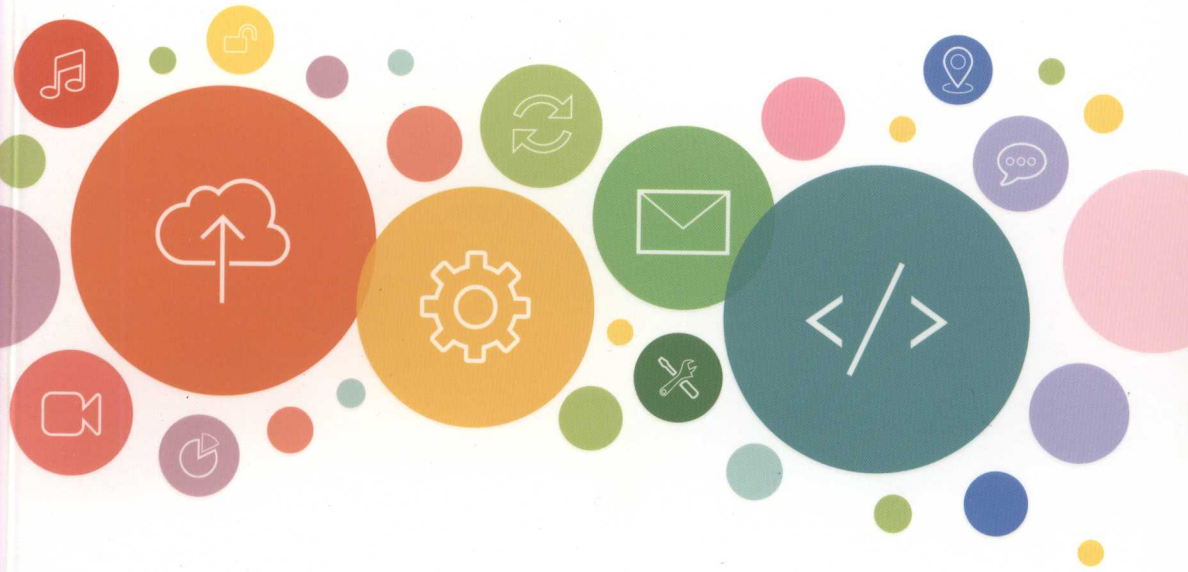


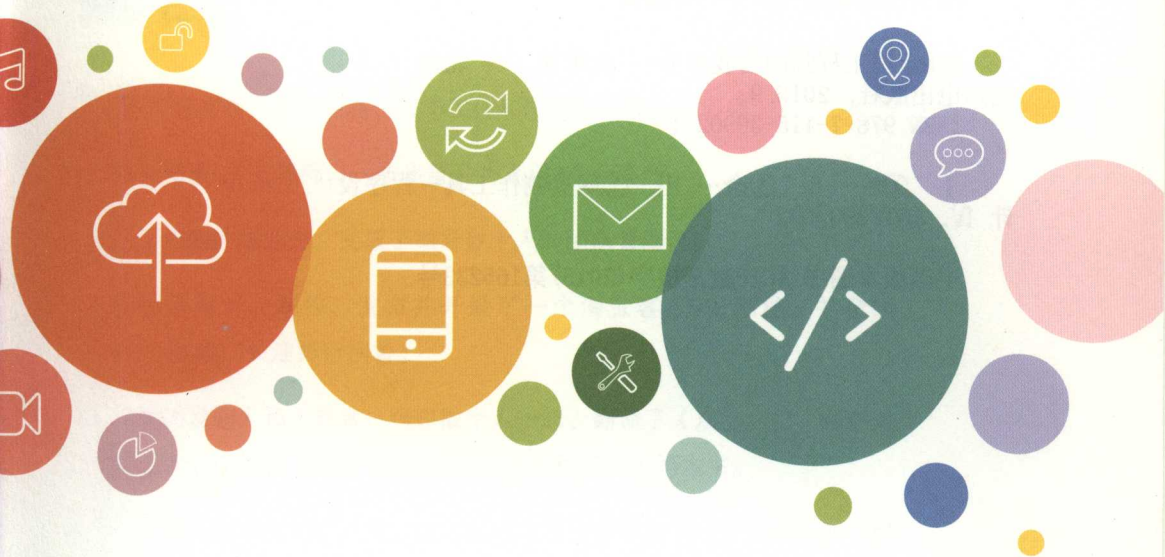
Web全栈工程师的自我修养

余果◎著

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

余果

腾讯社交用户体验设计部高级UI工程师，前端开发组负责人，熟悉前端开发、iOS开发、PHP开发和Ruby开发等；曾独立开发iOS APP（撸大师）和CMS（33PU）；翻译有《众妙之门：网站重新设计之道》和《响应式Web设计全流程解析》；平时喜欢编程、写作、演讲、摄影和英语等，希望自己能做一个终生学习者。



Web全栈工程师的自我修养

余果◎著

人民邮电出版社
北京

图书在版编目（CIP）数据

Web全栈工程师的自我修养 / 余果著. -- 北京 : 人民邮电出版社, 2015.9
ISBN 978-7-115-39902-1

Ⅰ. ①W… Ⅱ. ①余… Ⅲ. ①网页制作工具－程序设计 Ⅳ. ①TP393.092

中国版本图书馆CIP数据核字(2015)第165233号

◆ 著　　余　果
责任编辑　赵　轩
责任印制　张佳莹　焦志炜
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京方嘉彩色印刷有限责任公司印刷
◆ 开本：720×960　1/16
印张：15
字数：340 千字　　2015 年 9 月第 1 版
印数：1－3 000 册　　2015 年 9 月北京第 1 次印刷

定价：49.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

前言

您手中的这本书，是我在腾讯五年工作和学习的一些个人心得。

- 我从助理 UI 工程师，一步步晋升为高级 UI 工程师。
- 我从稚嫩的毕业生，变成了领导数十人的团队管理者。
- 我独立设计、制作、发布并开源了一个淘宝客 CMS 系统，并登顶 GitHub 热门关注排行榜首。
- 我作为发起人和主导者，组织数十人一起，翻译了《众妙之门：网站重新设计之道》和《响应式 Web 设计全流程解析》两本书。
- 我从零开始学习 iOS 开发，半年后独立制作并发布了一个 iOS App，赚回了两年的开发者费用。
- 我从一个不敢对陌生人讲话的菜鸟，变成了在几百人面前分享的演讲者。

在这五年中，我最大的收获就是，领悟到做一个“全栈工程师”的快乐。能够做自己喜欢的事情，能够全心投入，能够边学边做，能够不追求完美，能够自我驱动，能够不被自己的头衔局限，能够看到不同技术的联系，能够被老板认可，能够被业界认可，能够相信自己……

由于平时的工作和技术学习都比较跨界，所以在几年前全栈工程师的话题刚刚兴起的时候，我就进行了很多研究和思考。哪些技术对一个组织是真正有用的？全栈工程师的标准能力模型是怎样的？为什么有些人学习和晋升更快？

带着这样的思考，从 2014 年开始，我在豆瓣网发表专栏《谈谈全栈工程师》，发表了 20 篇连载专栏之后，得到了很多读者的欢迎，有五千多人订阅了我的专栏，并且在评论中跟我交流心得、表达感谢。我在开心的同时，也知道自己写得还不够好，文章还有很多语法错误和逻辑不清的地方。于是我打算投入更多心力出一本更好的作品。

经过半年的整理和撰写，这本书终于完成了。我把这本书定义为“轻松的技术杂文集”，希望读者可以以轻松一点的心态来读。书中一小部分内容来

自豆瓣网专栏的扩充，一小部分来自我的博客（http://yuguo.us），一小部分来自这一年多来的梦境和灵感，一大部分想法来自阅读。

本书需要读者有基本的编程基础，能理解基本数据结构，了解一门编程语言的语法。

如果可能，本书尽量不提供某种具体语言的代码实现。此外，读者可能对某一章的内容想作深入的了解，因此我在每一章节的末尾提供了延伸阅读推荐。

关于我

简单说说我自己吧，我从小一直很喜欢读书，高中开始对计算机技术燃起狂热的兴趣。还记得高中时候我每个月必读的两本杂志是《大众软件》和《散文》，即使是最忙碌的高三也没有停止，毕业的时候杂志堆起来一米多高。

《大众软件》话题覆盖面很广，从游戏评测到硬件展览报导，从软件推荐到硬件速递，从手机评测到 CPU 架构介绍……也许从那个时候起，我就养成了对各种新技术来者不拒的习惯吧，这也是我下定决心报考 IT 专业的原因。

小学时候看过很多散文、唐诗宋词，这可能跟我父母都是文科生有关系；可我偏偏热爱并擅长理科，尤其数学和物理，长大后渐渐喜欢看编程类的书。在父母都是文科生的环境下长大，导致我可能有一种感性和理性相结合的特质。

从理性的角度来讲，我做事情非常在意逻辑、证据、数据和对比；从感性的角度讲，我喜欢把我理解的知识用图形化的方式储存在脑海中。还记得高中做数学题的时候，有些关于象限的题，既可以用方程和公式去计算，又可以用图形去推理，我就非常喜欢用图形去推理，看到一个方程式就能“脑补”出解的集合曲线。一个方程组的解就是象限图种几条曲线的交集，因为线是点的集合，所以交点就是既满足方程 A 也满足方程 B 的解，这对我来说是非常容易理解的事情。但副作用是，由于长期不开发自己的记忆能力，所以我很容易忘事，也经常记不住人的名字和脸。理科中，我的生物和化学成绩就不怎么好。

后来我在西安电子科技大学读软件工程专业，西安是一个很美的城市，在沙尘中有一种古老的沧桑感，整个城市也方方正正（处女座最爱），鞋子和衣服在

阳台上放两天就会有一层灰。我很喜欢西安，我在西安度过了美好的四年。

从大学第一天起我就开始写博客，大学生时间比较多，期间折腾了很多域名和很多服务器，以及各种各样的博客程序，也丢过很多内容。在大学毕业那年，我开始启用 `http://yuguo.us/` 这个域名，并抛弃 WordPress，开始用静态站点生成器 Jekyll 生成站点，并使用 GitHub Pages（https://pages.github.com/）提供的免费服务器来托管页面。使用静态页面的最大优点就是访问速度非常快，而且不会出现服务器错误和数据库错误。如果说各种博客程序之间的 PK 就像高手对决，那么 WordPress 这种重型 CMS（Content Management System，内容管理系统）就像降龙十八掌，变化多端，是力量和技巧的极致，有一种无法掩饰的王者霸气。Jekyll 则像是小李飞刀，不会与您正面争锋，但是“小李飞刀，例无虚发”，是速度和精准的极致。

因为非常喜欢折腾网站前端的技术，所以在毕业的时候，我意料之外而又情理之中地选择了前端工程师这一个职业，并且很幸运地在校园招聘中，初次面试腾讯就被录取，并在腾讯工作至今。后来证明，大学的软件工程专业学习很有用。读书时觉得理论知识和后端的知识比较无用，但在工作中却证实，它们非常重要，所以我现在也经常回头复习一些基础知识。

就像乔布斯在斯坦福大学那场著名的演讲里说的，一个人在年少的时候，可能无法看到自己现在做的事情跟自己的未来会有什么关联。您无法预知未来，只能回顾。但是您需要有信心，当您很多年后回头看时，这些点点滴滴会连接在一起，让您朝自己的理想迈进。

我无法预知未来，但回头看过去的五年，我在这期间遇到种种困难，解决各式各样的痛点，帮助项目和团队成长，并成就自己的成长。虽然这些痛点不是每个人都会遇到，世界上也没有完全相同的项目，但是我觉得全栈工程师的理念是通用的，所以我的经验可能对其他人也是有帮助的，这也是我写这本书的初衷。这些思考，我会在本书中一一道来。

最后我想说的是，做您自己感兴趣的事情，学您想学的知识，不要怕走偏了，如果有人说您不务正业，那就让他们说去吧。如果您能远离传统的路子，您将会不同凡响。

目录

什么是全栈工程师

全栈工程师（Full-Stack Engineer），是一个在 IT 行业圈子里越来越热门的话题，无论是像 Facebook 这样的大型公司，还是刚刚起步的初创公司，都开始招募全栈工程师。据说，Facebook 声称：“我们只招全栈工程师！”

Facebook 只招全栈工程师

“全栈”是一个外来词，对于中国读者而言，会觉得它很陌生。当我第一次对某人提到“全栈工程师”时，他一头雾水：“全栈？您是说全端工程师吗？”

其实，“全栈”翻译自英文 full-stack，表示为了完成一个项目，所需要的一系列技术的集合。“栈”是指一系列子模块的集合。这些软件子模块或者组件组合在一起即可实现既定功能，不再需要其他模块。

全栈中的“栈”与计算机数据结构中的“堆栈”不是同一个概念，后者是指先入后出的串行数据结构。顺便说下，“队列”是指先入先出的串行数据结构。

IT 行业之外的人其实很难理解 Web 开发是多么复杂的工程。人们一般认为，在计算机公司或者互联网公司工作的人，就应该能够解决与计算机相关的所有问题：电脑开不了机、应该买什么型号的手机、家里上不了网，等等。在他们眼中，计算机行业的从业者天生就带有“全栈光环”。

但是拿着这本书的您知道，要开发一个 Web 页面，工程师需要掌握的知识至少包括：服务器（比如 Linux）、数据库（比如 MySQL）、服务器端编程语言（比如 PHP）、前端标记语言和脚本语言（HTML、CSS、JavaScript）等。这些技术中的每一个，都需要几年的学习和练习才能达到精通的程度。Web 工程是一个如此大的专业类别，以至于 IT 公司为每一个环节都设置了专门的部门和岗位，来把每一个环节做好。

服务器、数据库、服务器端编程语言、HTML、CSS、JavaScript 等组合在一起就是一个“栈”。这个“栈”是用来制作 Web 站点的，所以又叫 Web 栈（Web-Stack）。[1]

1 最常使用的服务器是基于 Linux 的。Web 发布使用 Apache，数据库使用 MySQL，服务器端编程语言使用 PHP 的组合，所以它们往往一起统称为 LAMP（Linux-Apache-MySQL-PHP）整体解决方案。

如果要开发一个在手机中运行的应用，开发者需要的知识包括：服务器、数据库、服务器端编程语言、iOS 或者 Android 开发技术。这些技术的集合称为 App 栈（App-Stack）。

一个简单的 Web 栈模型：包含前端技术和后端技术。

我们知道，前端工程师就是负责页面浏览器端编程的人，后端工程师就是负责服务器端编程的人，那么什么才是全栈工程师呢？

对于全栈工程师，业界并没有严格的定义，并不是说一定要一种都不能少地具备哪几项知识才能叫做全栈工程师。我倾向于认为，**应该从能力和思维方式两方面，来判定一个人是否是一个合格的全栈工程师。**

国外是怎么定义全栈工程师的呢？在著名的问答网站 Quora 上有人提出了这个问题。一个获得了高票的回答是：

> 全栈工程师是指，一个能处理数据库、服务器、系统工程和客户端的所有工作的工程师。根据项目的不同，客户需要的可能是移动栈、Web 栈，或者原生应用程序栈。

基本上，当客户需要一个全栈工程师的时候，客户需要的是一个**全能的“大神”**。简单来说，全栈工程师就是可以独立完成一个产品的人。当客户让他去做一些舒适区之外的工作时，他敢于迎难而上，并成功完成任务。

我们每一个工程师，进入到公司和企业工作之后，就会有一个职位头衔。我的职位头衔是“UI 工程师”，其他人的头衔可能是“交互设计师”“PHP 开发工程师”，等等。“全栈工程师”不需要头衔。他既有全面的技术能力，也渴望跨界工作的状态。

“全栈”好像是一个遥不可及的梦想，所以对于初次了解“全栈工程师”这个概念的工程师而言，有可能觉得“不可思议”或者抱着“这不可能”的排斥心理。但如果我们回头看看 Web 开发的历史，就知道“全栈”其实没那么难。

Web 开发流程

有人曾开玩笑说，全栈工程师是资本家的阴谋，因为老板想雇一个人来做三个人的工作。

其实在 2000 年第一次互联网泡沫破裂之前，那时候的 Web 工程师也许符合“全栈工程师”的简单定义：**一人包揽整个网站的构建**。

那时的 Web 工程师们所面临的挑战比今天小很多，他们可能只是制作一些静态的页面，不会面对如今富交互的 Web 应用程序。那时网站可能包含数据库和一些 HTML 表单，但仅此而已，甚至只需要将一些静态页发布到服务器上。在网站的前端无需视觉设计和交互设计，因为网站屈指可数，市场竞争很小，工程师仅用一些基本的 HTML 标签和闪亮的 GIF 图片就可以吸引网民的目光。同时，网站访问量都比较小，前端资源的体积也不大，无需关注服务器压力和 CDN，网民对加载速度的容忍度比较高，也不需要过多考虑用户体验。

但随着技术的发展、用户量的增加、客户端种类变多，每一个小小的细节都需要优化和考虑。在海量的访问量面前，也许改变一个按钮的位置和颜色就能影响上千万的订单。如今的互联网产品已不是以一己之力就可以完成的乐高积木了，Web 开发需要以某种可控的方式来管理。

于是，所有认真对待互联网产品的大公司都引入了流水线开发流程，在这条流水线上诞生了多个非常专业的职位。

大中型互联网公司的产品研发流水线。

产品经理：产品经理其实是对一个产品负根本责任的管理者。他通常的工作包括制订产品规划、协调多方资源、把控产品方向和质量细节，等等。有时候，他会从头策划一个新的产品，而更多的时候，他是在优化已有产品的一个部分。总之，在流水线中，产品经理需要从策划跟进到发布，是一个非常重要的角色。

用户研究员：用户研究员的工作是研究用户行为，有时候他会从宏观的角度分析数据，有时候也从微观的角度分解用户场景，有时候会召集一些用户专门来访谈，或者观察用户对产品的使用情况。从输出品的角度来说，用户研究员一般输出用户研究报告来交付给产品经理和交互设计师，作为产品设计的目标参考。

交互设计师：交互设计师常被简称为“交互”。他与视觉设计师最大的区别是，交互设计师更多着眼于如何优化用户界面的信息分布和操作流程。交互设计师的输出品一般是描述用户与网站“交互”过程的流程图，以及描述页面信息结构的线框图。输出的线框图会交付给视觉设计师。

视觉设计师：在细分交互设计师和视觉设计师的大公司，视觉设计师根据交互设计师输出的线框图来做一些润色和设计，输出最终的产品视觉稿之后将视觉稿交付给前端工程师。在一些不细分交互设计师和视觉设计师的小公司，二者被统称为“设计师”，他们的职责就是负责整个用户界面的设计。

前端工程师：产品视觉稿在得到产品经理和交互设计师等多方确认之后，会交给前端工程师，由前端工程师制作页面，实现视觉稿以及交互功能。从头衔上的变化就可以看出，这时候才真正开始编码。前端工程师需要非常熟悉 HTML、CSS 和 JavaScript，以及性能、语义化、多浏览器兼容、SEO、自动

化工具等广泛的知识。[1]

后台工程师：使用服务器编程语言，进行服务器功能的开发。在编程语言的选择上，很多公司都会出于团队已有成员的知识储备、程序员的供给量或者语言性能方面来进行选择。在这一方面，后台语言的选择是相对自由的一件事，不像前端工程师，为了页面兼容性，必须使用 HTML 和 CSS。如果关注各大公司招聘信息的话，您就会了解，不同公司使用不同的后台语言，比如传统的 C# 和 C++、Java、PHP，或者新潮的 RoR 和 Python。小公司的后台工程师除了负责功能开发，可能还会负责服务器的配置和调试、数据库的配置和管理等工作。在大公司，这些工作会分别委派给后台工程师、运维工程师、数据库管理员（DBA）等岗位。

运维工程师：运维工程师是跟服务器打交道的人，他会关注服务器的性能、压力、成本和安全等信息。

测试工程师：顾名思义，测试工程师保证产品的可用性，即使在小公司，这一职位也是不可或缺的。

流水线的优势

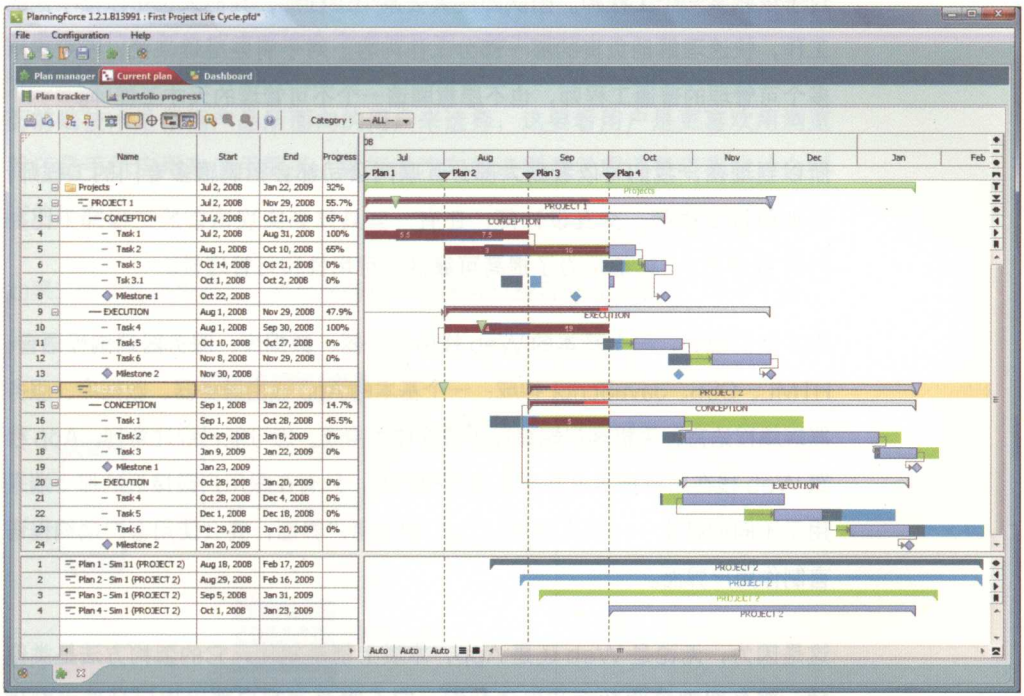
由于有了流水线，其中每个职位的可用工作时间都会作为“资源”来管理，因此需要一位项目经理来把控项目进度，并对人力资源进行调控。比如一个项目立项时，就要预约好这个周期版本需要实现哪些优先级较高的特性，而把优先级不那么高的特性推迟。对于确定在这一周期要实现的特性，就要安排本周进行设计、下周完成开发、下下周进行测试等。

在项目管理中，经常会用到甘特图。甘特图（Gantt Chart）是柱状图的一

1 对于前端工程师这个环节，腾讯公司进行了更进一步的分工，分为“UI 工程师”和“前台工程师”。UI 工程师主要负责 HTML 和 CSS，在制作的过程中要考虑非常精细的设计还原、语义化、页面性能和 SEO 等，而不用考虑 JavaScript 以及页面数据。前台工程师主要负责 JavaScript，他会在静态页面的基础上增加动态数据以及 JavaScript。不是所有的大公司都细分这一职位，在百度和阿里巴巴，前端工程师一个人同时负责页面还原和 JavaScript 开发。

种，显示项目、子项目、进度以及其他与时间相关的系统的进展情况。

流水线在大公司的任何一个严谨的大型项目里都是必不可少的，因为无论是 Web 产品还是 App 产品，它的复杂性都已经超出了单个工程师可以控制的程度。通过把复杂度分解到各个组件，每一个组件就可以进行很好的质量控制。



用于项目管理的甘特图。

Web 页面的生成和传递需要经历复杂的过程，因此容错能力就是首当其冲要考虑的问题。数据从位于深圳某个机房里的服务器传输到用户手机浏览器页面上进行运算和渲染，这个过程中的每个环节都可能出错，所以每一步都要做好容错处理。如果服务器出现错误，是否能在 30 秒内切换到备用机？后台数据异常时返回什么结果给前端，等等。

Web 页面可以在无数种设备上显示。兼容性在此时成为了前端工程师需要考虑的一个重要问题。不同的用户在不同手机上浏览页面，显示的方法会有些许不同，甚至要考虑到如果浏览器不支持 JavaScript，则需要给出特定的提示。

模块化的 Web 开发流程在很大程度上提高了服务的可靠性和可用性，让我们对每一个环节都能单独进行测试。这让大型 Web 开发真正变得可管理、可控制、质量可评估。

流水线带来的另外一个好处是，产品以团队的方式来运作和生产，公司不会过于依赖某一个工程师。团队即使失去某个工程师，其他人也可以接手他的工作，快速理解他负责的那一部分工作内容。对于有些经理来说，宁可雇用多个可管理的普通工程师，也不愿意聘请一个不可管理的天才工程师。

所以到现在，我们可以看到大部分互联网公司都会招聘很多专门的工程师，比如前端工程师、交互设计师，还有一些具体到实现语言的工程师（比如 PHP 程序员），这都是为了提高可靠性、可用性和可管理性。

刚才我们说到，一个基本的 Web 栈由服务器、数据库、服务器端编程语言、HTML、CSS、JavaScript 构成；一个基本的 App 栈由服务器、数据库、服务器端编程语言、手机客户端编程语言等技术构成。您可能已经注意到，App 栈跟 Web 栈在后台技术上几乎是完全相同的，只有在跟用户最接近的那一端采用了不同的技术——要么使用 HTML 制作用户界面，要么使用客户端编程语言制作用户界面。

这是因为，无论是 Web 还是 App，本质上都是软件，它的架构方法是类似的。服务器端接收数据和发送数据，它无需关注客户端采取何种技术制作用户界面。客户端处理用户交互以及显示数据，它不关心服务器使用的是 Java 还是 PHP。如果说开发一款软件就像制造一辆汽车，那么服务器端就像动力系统，客户端就像汽车的车身，不同的动力系统和车身可以自由组合搭配（我不太熟悉汽车的制造过程，这里只是作个比喻）。

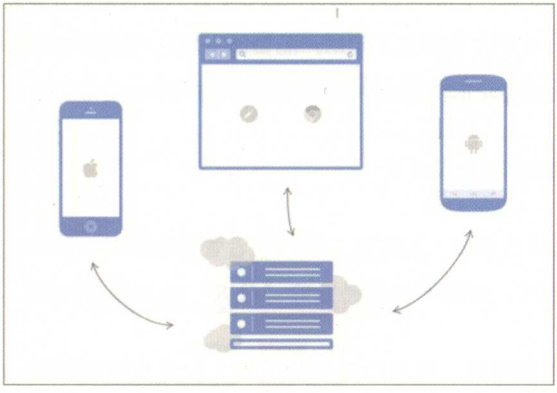
服务器和客户端之间通过 HTTP 协议传递信息。正是因为 HTTP 协议的通用性，使得服务器端和客户端得以实现完全的技术分离。无论是开发 Web 服务还是手机里运行的 App，一套后台开发技术，可以为所有的前端展现方式实现软件的商业逻辑。

HTTP 协议类似于汽车组装过程中的一个通用标准，动力系统和车身都要采用这个统一的标准来实现才可能完美对接。

用户量的大小、服务器承受压力的能力、软件对服务器计算量的要求、对服务器响应速度的要求……诸多因素会影响开发者决定使用哪一种后台技术。汽车的动力性能主要由发动机来决定，汽车厂商也会根据市场需求、消费者定位和制造成本等综合考虑使用哪一种发动机。

而前端技术是根据产品所面向的用户来选择，这要看用户是更喜欢用浏览器还是手机应用来使用服务。就好像汽车的造型要考虑消费者喜欢什么样式的外观。

如果二者功能分离得当，后台服务跟前台服务一般可以自由搭配，互不干涉。



如果服务器逻辑和客户端逻辑分离得当，二者可以自由搭配。

“各司其职”的弊端

虽然流水线式的职业划分和工程管理有很多优点，但是它就像一把双刃剑，在带来高可控性、可用性和可管理性的同时，也给工程师带来了一些困境。

工程师职责不清导致效率低

因为分工太细，所以在不同职业的交接处往往会有一些既不属于上游，也不属于下游的“灰色地带”。

这部分工作没有明确规定由谁去做，所以有时候时间会浪费在沟通上。员工会认为自己的头衔代表了自己的责任边界。比如，一个前端工程师可能会不加思考地实现视觉设计稿，因为他的岗位说明里规定了自己的职责，这其中不包括质疑设计稿，所以他忽视了自己的最终目标：让产品更好。

在一个开放平等的环境中，他实际上可以对影响可用性和性能的设计提出自己的想法。甚至如果他很熟悉这个项目的话，对设计的一致性和一些交互细节都可以说出自己的看法。

- **工程师缺乏主人感导致产品质量差**

流水线工作流程对专精工程师的要求是，能很好地执行动作或者执行任务，而不需要对产品的目标有很好的理解。其实在工程师的初级阶段，执行任务的能力是必需的，因为他还没有能力把握产品的目标，而且也需要更多的练习来提升专业能力。但随着经验的积累，如果工程师还不能对产品整体有自己的理解和贡献，就很容易缺乏主人感，要么他会跳槽，要么产品本身缺乏亮点而导致失败。

- **工程师缺乏全局的视野影响个人成长**

当工程师希望晋升到更高级的职位，如高级工程师或者管理岗位时，公司对他的大局观会有更高的要求，这就不仅仅是做好“分内”的工作就行的。

高级工程师需要有对设计的理解、对后台知识的了解，以及有跨团队推动项目的能力。长期研究专精的专业知识会让一个人视野变窄，变成“学术派”，而不是“实践派”。

- **更多角色导致项目效率低下**

软件工程项目与工业中的标准流程化项目有一个很大的区别：标准流程化项目中每一个流程所接受的输入都是一样的，所需要的输出也都是完全相同的。

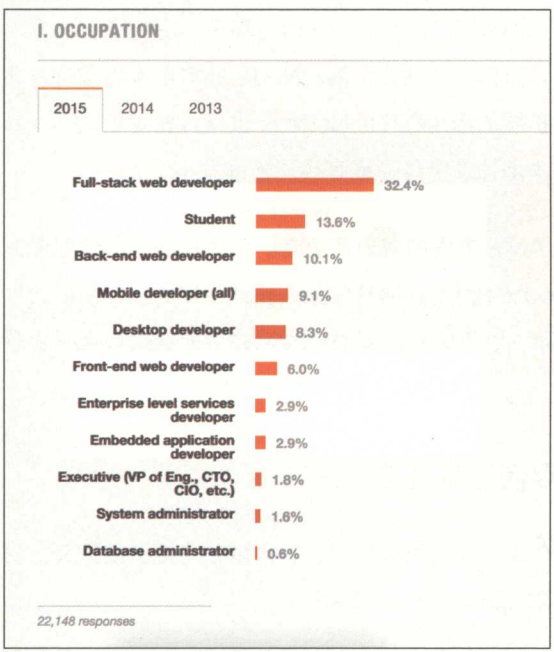
比如，一个汽车生产流水线，将“造汽车”这个任务分解成“造轮胎”“造方向盘”等。流程拆分得越细，每一个工人或者机器人就能做得越快，整

个流水线就会运转得越快。但是在软件工程项目中，我们把任务拆分给多个角色，每一个角色需要同样多的时间去理解需求，在上下游时间的安排中也往往会增加一些缓冲时间，比如周一安排设计，但是为了防止延期风险，会安排周三再制作前端页面。每一个角色的工作时间都会变长，而且交接也增加了缓冲时间，这样整个项目的时间就会被延长。

全栈工程师登上舞台

因为各司其职的工作流程有效率低下、成本高的缺点，所以很多创业公司都不会配备齐全的流水线，而是希望采用更灵活的方式来组建团队，全栈工程师也因此成为了理想的选择。但是全栈工程师的兴起还离不开这两个重要因素：技术的发展，以及提供 PaaS 服务的平台越来越多。

根据 StackOverflow 在 2015 年进行的开发者问卷调查，**有 32.4% 的开发者是全栈工程师，这一比例连续三年来逐年上升。**



您是哪一种类型的开发者？截图来自 2015 年 StackOverflow 开发者调查。

技术的发展

提到全栈技术，不得不提一个代表性的全栈框架——MEAN，它是MongoDB-Express-AngularJs-Node.js 的缩写，是从数据库、服务器到前端页面的一个完整技术栈。

MongoDB 是一个面向文档的、NoSQL 类型的数据库。MongoDB 颠覆了传统的基于表的数据存储方式，而采取了类似 JSON 的文档结构来存储数据，因而它在储存数据时可以更加灵活。

Express 是一个 Node.js 框架，可以创建灵活的 Web 服务，比如单页面应用程序、多页面应用程序和混合型 App。

AngularJS 是一个开源的 JavaScript 框架，由 Google 和开源社区共同维护，它用来创建单页面应用程序。它的目标是使用 model-view-controller 模式来规范 Web 应用程序，让开发和测试富交互的单页面应用程序变得更加轻松。

Node.js 是一个运行在服务器端的 JavaScript 运行环境，它的底层是基于 Chrome 的 JavaScript 运行环境——V8 引擎。Node.js 可以作为服务器端语言，用来创建快速、可扩展的应用程序。Node.js 也可以在本机运行，做一些本地操作，比如加速本地开发流程，或者实现一键发布。

MEAN 可以说是传统的 LAMP 方案的有力竞争者。因为从服务器端到页面端都采用同样的语言（JavaScript）和同样的架构模式（MVC），所以一个擅长 JavaScript 的工程师可以兼顾前后端的开发，并且前端模板代码和后台模板代码是可以复用的。

提供 PaaS 服务的平台越来越多

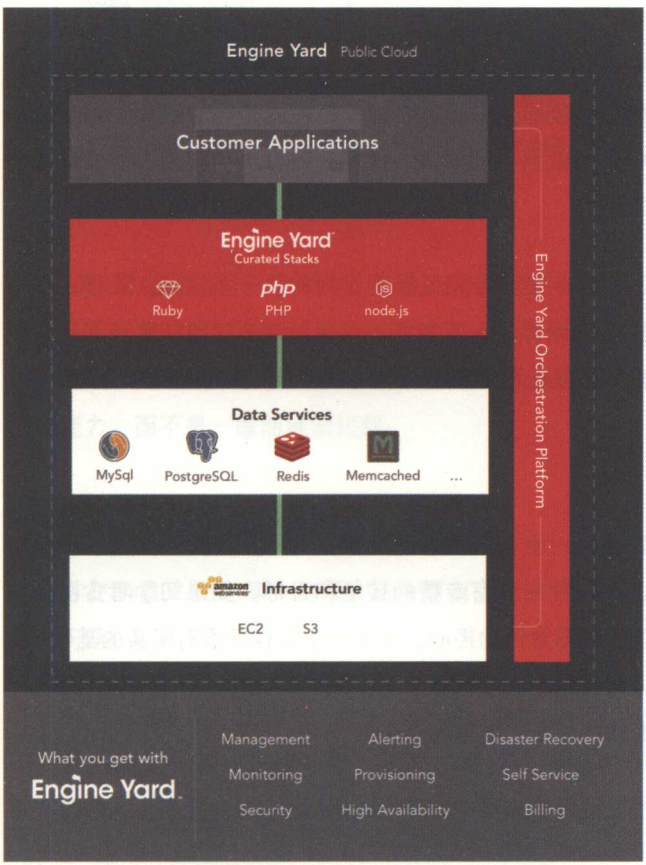
随着 Web 技术的发展和开源社区的积极努力，有很多公司提供便宜又方便的一条龙服务，可以解决独立开发者的大量麻烦。

比如 Amazon 提供的 PaaS（Platform as a Service，平台即服务），就可以

让创业公司的开发者省去架设和维护服务器的麻烦。

而 GitHub 在 2012 年获得了一亿美元融资，也可以看出市场对代码托管市场的信心。可以预期，未来可能会出现越来越多为开发者提供服务的公司。以后，小公司也可以用更低廉的价格获得世界级的 IT 服务支持，毫无疑问，更多的 IT 服务将托管在第三方的服务器上。

VPS（Virtual Private Server，虚拟专用服务器）是把一台物理服务器虚拟成多个虚拟专用服务器的服务。每个 VPS 都可分配独立的公网 IP 地址，运行独立的操作系统，拥有独立的磁盘空间、内存、CPU 资源、进程和系统配置，模拟出“独占”使用计算资源的体验。



EngineYard 为开发者提供一条龙服务（截图来自 www.engineyard.com）。

全栈工程师的发展前景

纪录片“寿司之神”讲述了86岁的顶级寿司制作者小野二郎的故事。小野二郎心怀追求极致的匠人心态，终其一生，他都在握寿司，永远以最高标准要求自己跟学徒，观察客人的用餐状况，微调寿司，确保客人享受到究极美味。他的寿司店只有10个座位，上厕所都需要去店外的公共场所，但是这样一间小店获得了米其林三星的顶级评价，这意味着仅仅为了享受美食就专程来到这个国家都是值得的。我想小野二郎就是专精工程师的代表，他日复一日地磨练和提高自己的技艺，他不会想要上市或者在全国开满连锁店，也不去追逐更大的商业回报，只为了自己内心对完美的追求。

确实，全栈工程师不是唯一成功的方式，也不是所有工程师的最终归宿。无论您是渴求成就感，还是物质回报，都有很多路径可以达到。如果能在任何专精的职业中努力做到名列前茅，就能获得巨大的回报，就像顶级的寿司制作者小野二郎。

而我推崇的全栈工程师则是与专精工程师不同的另一条道路。全栈工程师除了在一个专精知识领域有深入研究之外，还以知识广博和解决问题能力强著称。所以我认为有志成为全栈工程师的学习者，要有这样几个觉悟。

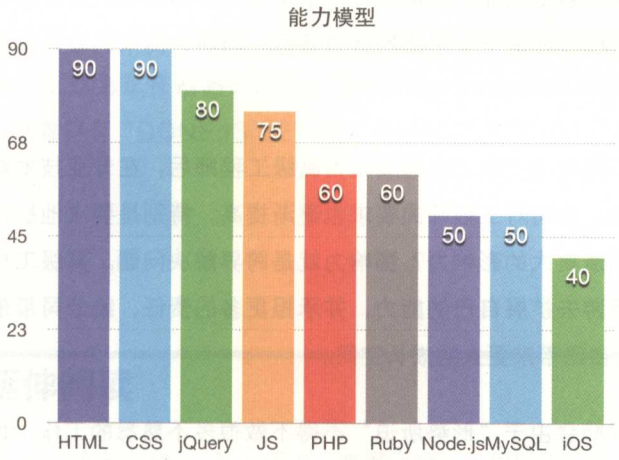
一专多长

我跟一位行业专家讨论过全栈工程师的话题，他不是很赞同全栈工程师这个方向。他认为，工程师应该有专精的技能和目标，如果初学者贪图大而全，反而样样不精。我理解他的担心，如果一个工程师没有坚实的基础（比如专业理论知识，对常用设计模式的理解，或者特定职业的基础知识），那么了解的非本专业技能越多，越容易迷失。

所以我认为，全栈工程师首先要“一专多长”。

一专多长的意思是，工程师首先有一个专精的方向，在这个方向上足够精

通之后（高级工程师级别），以此为突破点去学习更多的知识，增加自己的长处。如果还没有获得某个方向上足够深入的理解，就不要囫囵吞枣地去学习其他领域的知识。



全栈工程师最好“一专多长”。

有些知识需要时间的积累，并不是快速阅读就可以掌握的。“全栈工程师”这个名词可能会引起读者的误解。勿在浮沙筑高台，“全栈”是一个长期积累的过程，是专精型工程师在不断解决问题的过程中积累知识和经验所形成的能力，而不是一蹴而就的过程。

解决问题，而不是醉心技术

公司存在的意义就是解决问题，公司要解决用户的问题，而员工要解决公司的问题。

公司的问题可能是降低成本、扩大用户群、增加成交量、优化性能，等等。不同的问题优先级不一样，投入同样的时间，有的项目能为公司增加上百万的收入，而有的项目却只能增加几万。

互联网领域发展很快，问题的优先级永远都是在动态变化的，所以团队往

往每半年或者三个月就要回顾一下当前形势，并制定新的工作计划。如果新计划不是您擅长的，怎么办？您应该马上开始学习新的技术，这就是我说的关注问题，而不是醉心技术。

无论个人的目标和兴趣是创业，还是单纯希望学习更多的技术，或者学习项目管理，全栈工程师都是一个不错的努力目标。而随之而来的收益也是非常大的。

在大公司，程序员逐渐由初级工程师成长为高级工程师后，在专业技术能力上不断接近极限，公司对工程师的要求也逐渐提高，特别是要求他扩大“影响力”。如何创造更大的影响力？影响力就是跨界解决问题。**高级工程师可以选择往上下游去扩展自己的能力，并承担更多的责任，给公司带来更大的收益，也给自己带来更大的成长空间。**

在创业公司做工程师会由于“形势所迫”不得不做很多不熟悉的工作，也就是“舒适区之外”的工作。虽然说小公司没钱聘请更多员工是最直接的外在因素，但是全栈工程师的成长并不是靠外力，而是自我驱动。**程序员在小公司里主动去承担更多责任，自己跟公司都会获得相应的成长。**假如公司上市扩张，自己能获得巨额的回报，即使公司失败，自己也能获得锻炼。

在自由职业市场，全栈工程师是最闪耀的明星。因为全栈工程师能独立创作产品，所以很容易被市场接纳。比如 WordPress 主题设计、App 开发、网站开发，等等。全栈工程师也能轻松搭建自己的作品网站，而不像后台工程师的作品那样，不太容易展示的后台组件。

全栈工程师还是天生的创业者，因为自己可以独立完成一个产品模型，所以可以用最快的速度去测试自己的想法。从工作中锻炼出来的发现问题、洞察需求、设计解决方案并开发出初始版本产品的能力，是全栈工程师最大的优势。也许这就是为什么有些创业者说“我们就差一个程序员了”的时候，程序员们都会把他当成一个笑话。

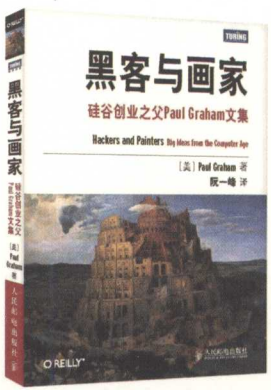
总之，全栈工程师是一个能够在所有场合发光发热、实现个人价值的职业。在未来，中国也会涌现出越来越多优秀的全栈工程师。

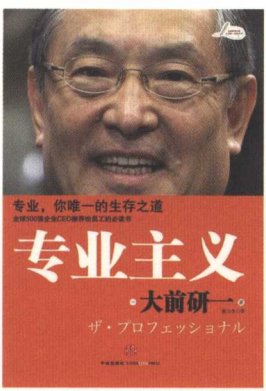
最后，关于上文中提到的“Facebook 只招全栈工程师”是一条谣传。从公司文化的角度来讲，Facebook 鼓励员工以开放的心态解决问题，不因为自己的头衔给自己设限。他们仍然招聘各种职位，您可以在官方网站上找到正在招聘的职位。我在 Facebook 参观的时候，看到一面墙上贴着写有“DONE IS BETTER THAN PERFECT”[1] 的海报，在此与君共勉。

延伸阅读

1. 《黑客与画家》（美）保罗·格雷厄姆，人民邮电出版社

2. 《专业主义》（日）大前研一，中信出版社





1 翻译为：完成比完美更重要。

如何成为全栈工程师

不管您是否承认，除去极少数天赋异禀、骨骼惊奇的天才程序员，我们大部分人都是普通人，都需要遵循“一万小时定律”，才能从平凡变成超凡。

凡人要从一个小菜鸟成长为全栈工程师，只能从少到多、慢慢积累知识和经验。职业生涯的本质，就是在一个专业方向上积累信息。这里我推荐采用“先精后广，一专多长”的流程来学习。采用这种方式来学习，不光可以触类旁通、举一反三，还让我们学习得更快，而且循序渐进更符合一般人的职业生涯发展。

先精后广，一专多长

围绕商业目标

关注用户体验

先精后广，一专多长

“先精后广，一专多长”是指，建议初学者学习全栈技能的时候，**先在一个特定的方向上有比较深入的钻研，然后再将学习目标渐渐推广开来。**比如先从前端方向入手，掌握了基本的 HTML、CSS、JavaScript 之后，不要转头向服务器端语言或者 App 方向发展，而是深入到性能优化、SEO、多种框架、响应式页面等前端细节中去。经过一到两年的深入研究之后，再去学习其他方向。

如果在创业公司做全栈的工作，一般也不会要求一个人处理所有的技术工作，至少会有两三个人组成团队来做项目。大家在分配工作的时候，可以按照每个人的偏好和技术特点，进行前后端的分工，不用完全按照每个人做一个模块的方式来分工。这种分工的界限不一定要很绝对，在不同职位的工作范畴中，可以有一些重合的区域。

如果是毕业生或者初学者，我不建议在刚开始的一到两年接触太多技术，杂而不精，结果可能会对后面的职业道路产生副作用。

为什么我强调在开始的时候有一个专精方向的重要性呢？因为这样您才能在求职的时候有一个“亮点”。

平心而论，程序员在市场上的供求关系比很多其他职业都更有利于求职者，在微博、Twitter、V2EX 上都会有很多引人注目的招聘启示，大家对优秀程序员的需求从来就没有减少过。

虽然优秀的程序员总是能找到工作并且工资不低，但是很多程序员投出的简历都石沉大海，一个主要原因是由于求职者的简历没有亮点，或者说从工作经历中提取不出来一个亮点。

让我们做一个情景假设，作为一个有两年工作经验的全栈工程师，您看到腾讯有一个职位空缺。

腾讯社交用户体验设计部招聘前端开发，要求如下。

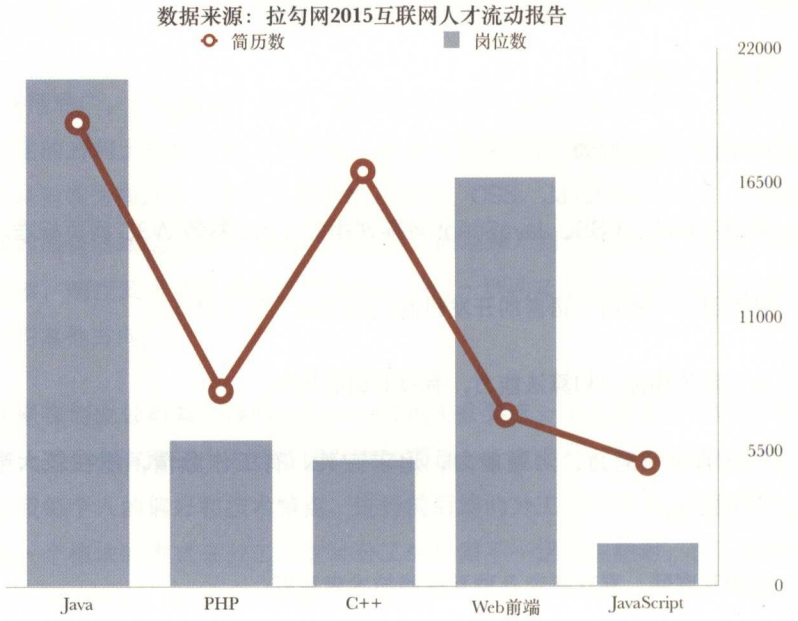
- 本科以上学历。
- 两年以上工作经验。
- 精通 HTML、CSS、JavaScript 等前端相关技术，熟悉 W3C 网页标准。
- 熟悉至少一种后台语言的开发机制（如 Java、C++ 等）。
- 有一定架构能力和算法能力，有良好编码规范。
- 良好的学习能力、沟通能力，追求完美，有工作激情，能在较大强度下工作。
- 热爱互联网，喜欢研究各种互联网技术者更好。

您想，自己完全满足要求啊，于是一封简历就投递到面试官的邮箱，里面用大段文字表达自己全面的能力完全符合这个要求，而且自己还有亢奋的激情和浓厚的兴趣。

但是您从面试官的角度来想想，他收到了多少份简历呢？对于一个大公司的 HR，可能 100 个都算少。

根据中国招聘平台拉勾网“2015 年互联网人才流动报告”，前端相关岗位的简历投递数只有岗位数的一半。与此同时，服务器开发方向（比如 Java、PHP、C++ 等）的简历投递数都大大高于岗位数。从图表可以看出，前端开发仍然处于人才紧缺阶段。

HR 要从 100 个符合要求的人中选择 10 个来面试，您的简历中的哪一点能吸引他呢？有的竞争者有丰富的移动端作品，有的竞争者提到他很擅长页面性能优化、响应式、页面渲染效率，有的写过 JavaScript 框架……而您只是一个普通的满足要求的人。



不同职位的供求关系是不一样的。

您可能会说，我爱好广泛，学习能力强，我会一点 PHP，做过 Wordpress 主题，会一点 Java，毕业设计做过一个小客户端应用，什么都会一点……但最终您仍然会得到一个“无亮点”的评价，被无情地淘汰掉。因为虽然您会的技能很多，但大多只能算是“及格”的东西。

所以，作为一个求职者，无论是毕业生还是社会招聘，仅仅满足招聘要求是不够的。**您需要在招聘要求的方向上以 200% 的能力来得到这个职位。**

一个求职者在整个流程中会受到多方考核：HR 考核您的成本和价值，专业面试官（不是全栈工程师）考核您的专业能力，经理考核您的沟通能力。在所有这些考核中，其实每一环都是漏斗型筛选，会过滤掉一些人。

好消息是，由于程序员的供求关系，只要您的专业能力够强，您就有很大的概率通过整个面试录用流程。我一次又一次提到“供求关系”这个词，是因为在商业社会，所有的商品（包括人才）的价值来自于供求关系，而

不是生产成本。生产成本是准入门槛，但绝不是核心竞争力。

让我再次重复这一点，作为求职者，一定要在某个特定方向上有非常深入的理解。仅仅会做还不够，还要理解背后的原因，还有背后的背后的原因。有些面试官的习惯是，在一个问题上深入地问下去，从经验问到操作过程，再问到技术原理，一直深入到面试官问不下去了，或者求职者答不上来了。所以，理解得越深刻，您就越有优势。

有了一个专长，得到一个能让您成长的工作，进入强大的团队，您就能有自己的阵地，以此为生，然后再逐步学习更加广博的知识，朝自己的个人目标去努力。如果您连阵地都不稳固，就不存在开枝散叶、落地生根的可能性了。

假设您已经在一个中等规模以上的公司找到了工作，那就会有一个专门的岗业。经过几年的工作和练习，您会在专业知识上达到很熟练的程度，日常需求都已经在您的“舒适区”，现在您终于准备好了。既然您的目标是做一个全栈工程师，那么从哪些技术开始入手呢?

围绕商业目标

我的第一条建议是，在考虑做什么项目的时候，围绕商业利益作为目标。归根结底，技术是服务于商业目标的。

在计算机科学诞生的短短几十年中，热门的技术和平台一直在发生巨大的变化。

服务器端的平台和语言从 C 到 C++、Java、Python，再到如今的 Node.js，变化从来没有停止过。

客户端则分浏览器和原生开发两个分支。浏览器方面，Web 标准是一个活的标准，意思是说，有一些新的提案不停地加入到标准之中，这是一个动态滚动的标准，而不是印刷出来的定案。

各种浏览器的市场份额每隔两年就会发生天翻地覆的变化，从 moz 到 Webkit，我们见证了 Webkit 的发展壮大。

移动端设备的市场份额之争更是激烈，曾经的诺基亚和摩托罗拉被新起之秀收购，iOS 和 Android 之争还在继续……

仅仅据我所知，2014 年到 2015 年腾讯就有很多团队进行了从 PC 端到移动端、从 HTML5 到原生 App 开发的各种转型。没有人能说得准下个季度我们团队的目标是什么，每半年就有一次大的调整，而小的调整从来就没有停止过。“变化”是唯一保持不变的东西，每个人都在不停地学习新的技术。

相对来说，商业目标是稳定的。把关注点放在商业目标而不是技术上，就能选择出更适合完成商业目标的技术，这样就能做出更为客观的决定。更重要的是，在这个过程中您学习到的不仅仅是技术，更是一种潜在的思维方式，这种思维方式可以帮助您提升综合竞争力，是一种“硬通货”的能力。

老板雇用一个员工，不是因为他能写程序，而是因为他能帮助自己赚钱。赚钱有两种方法：减少成本，或者增加收入。程序员如果能加快内部系统的运行效率，让产品制作流程更加顺畅，就是减少成本。如果能让用户更容易地购买产品，或者提高服务质量吸引更多用户，就能增加收入。在老板看来，程序员只是一个昂贵的劳动力，他会不会写程序都没那么重要，重要的是能赚钱。

所以如果您想成为一个高级开发者（或者高级设计师），就一定要学会这种思维方式。

所谓“商业目标”要广义地去解读。对于直接制作产品，给用户使用的团队，就需要对外关注如何提高产品质量、降低产品成本；对内应该关注如何优化流程、减少错误率。如果团队输出的成果是公司内其他部门需要的原材料，就要关注下游的需求，研究如何更好地输出成果，如何在流程上使得输出产品的过程更顺畅。

关注商业目标需要持久的练习。等到自己成为全栈工程师，或者成为团队

管理者，更加需要在多个目标任务之中做出选择。全栈工程师需要做和能够做的事情是很多的，他会很多技能，也负责处理很多工作，所以他更需要能力从诸多事情中找到最有商业价值的一个：可能是制作一款工具提升团队效率，也可能是成本上的优化。

全栈工程师可以做得事情越多，就越需要具备判断做什么的能力。如果增加一个用户需要的功能是加分项的话，拒绝一个用户不需要的需求更加值得推崇。

一切都要围绕商业目标来进行，包括您做的项目、您的汇报方式，以及您在学习新技能时进行的取舍。

我在公司的技术通道[1]中会发现有这样一些开发者，他们做项目的驱动力是“技术”本身，而不是“商业”目标。比如说，他们针对微信平台做了一个活动推广页，使用了很多华丽的 3D 旋转和 SVG 动画。好的方面如下。

- 用的技术很新潮，满足了自己的炫技虚荣心。
- 朋友圈（其实都是前端同事）传播很广。
- 在高端机器和大屏幕机器上效果很好。

坏的方面如下。

- 在低端机和慢速网络下效果不好。
- 沉浸在技术的实现中，而忽略用户体验。
- 打开页面就自动播放音乐，让用户感觉很突然。

我老婆是一位财务人员，她每次看到朋友圈这种很炫酷但需要加载的页面就会马上关掉，有时候耐心等待打开之后也是眼花缭乱，不知所以。所以有时候我会思考，一个技术的圈子，在热烈讨论某个推广页又用了某某炫

1 腾讯员工可以根据自己的特长和兴趣，选择走管理的发展通道，也可以选择技术、设计、产品、市场等专业发展通道。在不同的发展等级上，都设计有配套的能力要素。

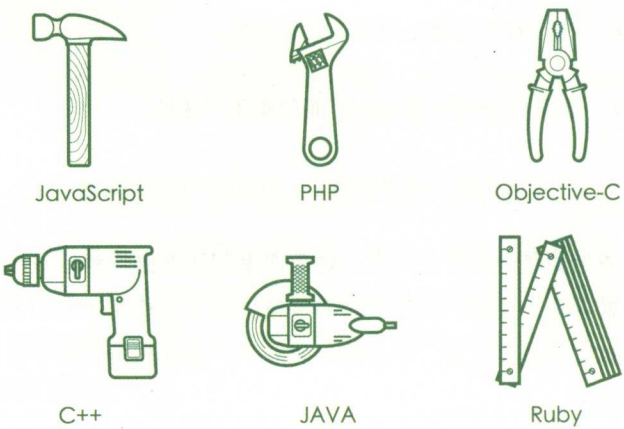
酷新技术的时候，有没有想到普通用户根本不买单呢?

再来说说一个好的案例。

我在面试求职者时遇到一个综合能力不错的候选人，他是一个全栈工程师。我问他，您现在掌握的技术比较多，那您未来的职业规划是怎样的? 他说，他觉得用什么语言并不重要，但是最近一年开始把重心放在 Android 开发上，因为移动端 App 开发是现在的潮流，有很大的需求，在这里可以有所成就。但在未来，不排除改变方向去做别的事情的可能，到时候可能是 iOS 或者其他新的系统。基本上来说，自己掌握的知识体系是可以复用的，但也期待学习新的语言。

我喜欢他这样的态度，对未来有自己的方向，但也知道自己没法看得太清晰。**对商业和市场有想法，而且自己也有足够的技术能力和自信向未来前进。**相比而言，有些候选者的项目经验和学习技能很杂，东一锤子西一榔头，有些时候纯粹是为了折腾而折腾。

记住，当您只有一把锤子，您看什么都是钉子。而如果您痴迷于工具，反而看不到问题所在。因此，要先看看有哪些问题需要解决，然后再补充您的工具箱。永远从商业目标的角度来决定学习哪些东西，而不是纯粹为了锻炼技术能力而去学习。



全栈工程师希望丰富自己的工具箱，而不是用一把锤子处理所有工作。

关注用户体验

我的第二条建议是，从用户体验的角度考虑问题。

用户体验是用户使用产品时的心理、感受、印象、评价。生活中处处涉及用户的体验，闹钟、牙刷、马桶、书包、公交、红绿灯、手机、电脑、键盘、鼠标……**每天，我们都在和产品打交道，每天都在使用和体验产品**。

每一个糟糕的体验背后都蕴含着商机

全栈工程师应该关注用户体验，并且掌握用户体验相关的知识。即使一个技术达人能够以一己之力搭建一个站点，但他如果不关注用户和客户的体验，那么他做的产品可能会很糟糕。这样的产品除了“能用”之外什么优点都没有。

所有优秀的工程师所做的一切都是在优化用户体验：优化性能的开发者是在积极地提升用户体验和交互；设计师有意用颜色、空间、大小和表单的排列方式让用户体验更顺畅好用；而内容运营者认为某些内容重要，某些内容不重要，也是在考虑如何提升用户的体验。

我在 2010 年加入腾讯的时候，公司只有一万多人。那时候，我需要办一些行政手续，需要公司开具薪酬证明，整个操作流程是这样的。

打开公司论坛，搜索“薪酬证明”，搜索到一篇文章，里面讲到找一位人力资源的员工来办理。

我打开 RTX（腾讯内部使用的工作通讯软件，类似 QQ），找到这位人力资源的员工，问他座位在哪里；跑到他座位上，此时已经有几个人在排队了，我排在后面；到我了，我告诉他我要办薪酬证明，并告诉他我的 RTX ID；等待 10 分钟后，他打印出一张薪酬证明，签字盖章后给我。整个过程耗费了我一个小时的时间。

2015 年，我要申请美国旅游签证，需要开具薪酬证明。我从平时的宣传渠道得知，现在人力资源的很多服务都可以在线上办理了，于是我尝试了一下，现在开薪酬证明的流程是这样的。

关注“HR 助手”的微信公共账号，它自动识别出我是腾讯员工，也得到了我的 ID。

选择“我要办证明”→“收入证明”，在证明用途一栏，选择“签证类”→“旅游签证”，并提交一些个人信息。

输入我的办公座位号，提交给系统。

第二天，一个漂亮的红色大信封放在我座位上。打开一看，里面包括中英文两份收入证明，还有我的旅游目的地以及时间，整个收入证明既漂亮又专业，是为签证量身打造。从提交系统到拿到最终的证明，我只花了几分钟，过程顺畅快速，体验非常好。

腾讯“线上申请”业务的体验极佳。

从 2010 年到 2015 年，经过这几年的发展，腾讯的员工规模已经达到了三万多人，翻了三倍。HR 流程如果还以旧的方式运作，可能得加派好几倍的人手，浪费所有员工不知道多少时间。但是现在通过自动化的系统，用户满意度大大提高。一个内部员工使用的系统，尚且有如此的优化空间和投入力度，何况是对外直接出售服务的公司呢？

我这样被公司服务“惯坏”的人，往往对社会上其他服务更加挑剔。此外，在深圳这样一个服务业水平居全国前三的城市居住惯了，去其他城市也经常会有被“怠慢”的感觉。我想这就是所谓“由俭入奢易，由奢入俭难”。

所以，用户现在都被手机中那些提供优质体验的 App“惯坏”了，想让他们再接受陈旧的设计和体验，就更加难上加难了。

用户是谁

“站在用户的角度想问题”这样一句朴实的话，可以指导我们做很多事情，但是很多时候我们忽略了这一步。

就像“体验”泛指所有生活中所有的体验。**这里的“用户”仍然是一个广义的定义：所有您为之服务的人。**

比如做一次演讲或者汇报，第一件要紧的事不应该是做 PPT，而应该是调查听众，站在听众的角度去思考：听众知道什么信息，听众想知道什么。如果给您的老板汇报，您不能期望他了解您所做项目的技术细节，而且他想知道的也不是技术细节，而是项目进度和风险。但是如果在一个技术论坛上分享，您就不能期望听众都知道您的项目背景和目标，需要花一点时间去介绍，听众也不想知道太多细节的东西，只需要介绍一些决策和架构的大方向。

写邮件的时候，收件人（还有可能这封邮件被转发之后的收件人）就是用户，那么写邮件的一些技巧就包括：尽量简短，不要给收件人太大压力；把结论放在邮件的开始，方便对方快速了解情况；如果需要老板拍板，给

出选择题，而不是问答题。总而言之，以对方能理解、会关注的方式来表达自己做了什么。

作为前端工程师，上游的设计师、下游的后台工程师，都可以认为是前端团队的用户。如果细心观察，就可以发现这里面有一些痛点。因为领导没有自己敲代码，所以他可能不会发现这些痛点，也就不会安排您去做优化工作。所以这里需要您自己去观察和优化流程。

很多程序员的第一个想法是做工具，但是想想我刚才说的话，老板雇用您不是因为您能写代码（或者做工具），而是因为您能帮他赚钱。所以您要用一切办法，去优化流程解决痛点，做工具是一个可选的方法，但不应该是您的第一个想法，更不是唯一的办法。假使真的是做了一个工具，最终汇报邮件的时候，不要以“我做了一个工具……”开头，而应该以“我发现了一个问题……”开始。

大巧若拙

老子（两个字都请以三声阅读）说，大巧若拙。

意思是，指**真正聪明的人，不会显露自己，反面从表面看好像还很笨拙**。用户体验不只是界面和交互这样可以直观感受的东西，还包括一些隐藏在用户界面背后的细节和规范。

就像冰山，露出水面的部分只占整个冰山的 1/9，用户看到的只是显露出来的部分。背后的部分一般用户是看不到的：比如用户研究，用研团队会通过调查，输出一些用户画像，影响整个产品的功能方向、设计风格；还有设计规范，设计团队在设计产品的一开始制定了规范之后，新增加的功能和页面都必须遵循已有的设计规范，这样整个产品是统一的，能够给用户专业的感觉。

为什么现在很多商业公司花了大把的钱和精力开发出独立运行的 App，体验却很糟糕，甚至很多用户反馈称 App 还不如微信公共号好用？

用户体验只是冰山上露出的一小部分。

一个很大的原因就是公司不重视用户体验，觉得用户研究和交互这种东西，不用专业人员去做，让设计师搞定就好了；或者老板拍脑袋定方案，做出的东西花里胡哨、炫酷狂拽，但就是让用户摸不着头脑。相反，微信花了很大的精力去做深入的研究，最后设计出了一套看似简单，但是可用性非常好的框架。然后微信开放后台系统给第三方，第三方公共号可以定制的地方有限，只能把功能往里面套，不太容易出错，用户体验自然就上来了。

反观某些银行的 App，几乎每个标签页的设计风格都不一样，而且喜欢自己设计键盘，每次在输入密码的时候都非常不方便，其实这是没有必要的。

做自己会用的产品

创业公司做产品，CEO 一定要是自己的目标用户。因为如果自己都不体验自己的产品，就很难发现用户在使用产品过程中遇到的糟糕体验。我们经常在网上看见网民抱怨办理公共事务时手续很麻烦，很多流程设置得让人抓狂。我想，这里面有一个很大的原因就是，设计公共事务流程的人，自己本身不是目标用户。

网上有个段子，说一般的产品经理没办法把自己代入成“小白”用户，做出的东西只有他自己会用；高级产品经理经过半小时的冥想可以进入小白状态；张小龙和马化腾这样的大师级产品经理需要两分钟；而乔布斯

可以随时切换大师级产品经理和小白的状态。这就是为什么他会说“stay hungry, stay foolish”。

我如果开创一个公司需要招聘“全栈工程师”，我要求的三个能力就是一专多长、关注商业目标、关注用户体验。

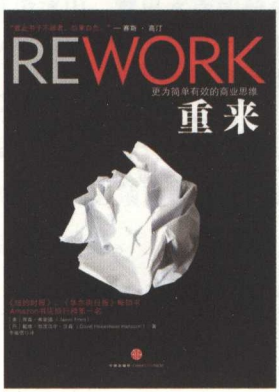
有志往全栈工程师方向发展的学生，我推荐从入门简单的前端开发开始学，而且从拉勾网“2015 互联网人才流动报告”来看，职位多、简历少排名第一的职位是前端开发。而且因为前端开发处于互联网开发的中间环节，可以从上下游入手，渐渐地接触 Web 开发的完整流程。第三个原因是，前端开发直接面对最终用户，也可以锻炼自己对用户体验的感觉。

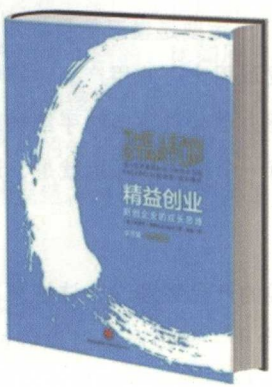
当然，前端并不是唯一的选择，您也可以从其他职位开始，专注地学习这个职位需要的技术，到达一定的深度之后，扩展自己的知识面，往一专多长方向去发展。下一章专门介绍如何从学生转型为全栈工程师。

延伸阅读

1. 《重来：更为简单有效的商业思维》(美)贾森·弗里德 /(丹)戴维·海涅迈尔·汉森，中信出版社

2. 《精益创业》(美)埃里克·莱斯，中信出版社





从学生到工程师

对于很多学生，临近毕业遇到的第一个问题就是，如何在一个城市找到一份工作。对于软件工程师来说，北上广深是最理想的选择。因为在大城市才有足够的工作机会，好的互联网公司都在一线城市，创业公司也都会选择人才和资金条件更好的北上广深。所以对于毕业生来说，要有在大城市工作的心理准备。

校园招聘

获得面试机会

实习

校园招聘

对企业来说，校园招聘是企业重要的招聘渠道。大学毕业生富有热情、学习能力强、善于接受新事物、头脑中的条条框框少、对未来抱有憧憬、可以全身心地投入到工作中。更为重要的是，他们是“白纸”一样的“职场小鲜肉”，可塑性极强，更容易接受公司的管理理念和文化。正是毕业生身上的这些特质，使得毕业生成为团队金字塔结构中重要的基础成员。

每年的招聘季，腾讯社交用户体验设计部都要招聘一些毕业生，我负责 UI 工程师岗位的初步人才选拔。我看过很多简历，也对其中很多人进行了电话面试和 QQ 面试，发现很多简历都是不得其所。

我想起当年自己的工作机会就是从校园招聘获得的，而且这几年也负责编写面试题、筛选简历、面试等招聘工作，有一些自己的经验。所以想跟大学生朋友谈谈我当年找工作的故事，也许能对学生读者有所帮助。

前端工程师要有一个基本常识，那就是结构、表现和行为要分离。技术层面上讲，网站的内容使用语义化的 HTML 标签，而不掺杂任何表现和逻辑；网站样式表现用 CSS 来描述，既能在多个页面之间复用，也可以根据不同用户来分别定义外观；页面行为逻辑用 JavaScript 来实现，这样保证浏览器在禁用 JavaScript 的时候，页面也能正常渲染和使用。

这些显而易见的基础知识，曾经不是那么显而易见。时光荏苒，岁月如梭，我的大学时代在西安电子科技大学度过，读的是软件工程专业。学校的课程基本上是一些编程原理、算法导论、数据库、操作系统，或者是 C 这样的很底层的东西。因为教 C 的老师不告诉我学好 C 能做什么，所以最终我对前端开发更感兴趣。我享受一个页面经过自己的调整而发生变化的过程。

当时 Web 标准刚刚开始在中国推广不久，但是学校的课程和图书馆的参考书里面，ASP、JSP 和 PHP 代码示例都不讲究这些，都是混合所有的前端代码和后端代码。而我开始自学 Web 标准后，知道了“结构、表现和行为分离”，也在这个方向上越钻越深。

校园招聘一般每年 9 月中旬就开始启动，主要集中在 9 至 11 月和次年的 3 至 4 月。9 月初，毕业生最后一个学年刚开始就进入了招聘季，出于招揽优秀人才的考虑，越来越多的企业都在越来越早地进入校园，通过校园宣讲会的形式提前介入到校园招聘活动中。10 月份则是目前校园招聘最繁忙的旺季，高潮会一直持续到 11 月底。春节前后则迎来了校园招聘的淡季；节后 3 至 4 月份会再现一次小高潮，主要争夺公务员考试和研究生考试失利的毕业生。

腾讯公司的校招页面。

2009 年 9 月，校园招聘的大潮刚刚开始，校园里热闹纷呈，板报区贴满了各大企业的招聘和宣讲海报，每隔几天就会被新的招聘海报刷新一遍。对于企业来说，在大学讲堂举办的各种宣讲一方面是密度最大的招聘广播，另一方面也是绝好的免费广告，会场里挤满了听企业讲企业文化的学生。

作为即将毕业的应届毕业生，我没有去听任何的宣讲会，也没有海投简历。相反，我跟那些准备考研的同学一起，每天泡在图书馆。这并不是因为我想考研，我觉得对自己而言，读研并不是一个很好的选择，我希望到企业中去实践。

那时候，我看了很多书，《禅意花园》《网页重构》《超越 CSS》《CSS

Mastery》等，我在酝酿一个计划。我的想法是做一个项目，来表达我所理解的 Web 标准，然后用这个项目来找工作。

经过了一些思考，最后我想做一个类似于“禅意花园”的静态站点，利用一套 HTML 代码写出我的简历，然后调用不同的 CSS 实现完全不同的设计方案。我还使用了当时很时髦的 AJAX 技术来动态切换样式，实现随机调用一个设计方案。在别人都在海投简历的时候，我就把精力全都放在这个项目上。没有人可以教我，软件学院没有这个方向，同学中也没有几个前端爱好者，我只好自己傻傻地摸索。

现在回过头来看，我当时并没有想到自己会进入腾讯。每天在路过图书馆和宿舍的路上，我也会看看路边的招聘广告，但我并不看公司，而是看前端工程师的职位。有一天我在海报栏看见腾讯的招聘，但是只有设计师、后台开发和产品经理的岗位，所以我就走过去了。

有些同学优先看公司，其次看岗位，能进入腾讯就行。**但我认为岗位优先于公司，即使在一个很好的公司里面，如果只是做着自己不喜欢也不擅长的工作，那能有什么前途呢。**

我以为我会与腾讯擦肩而过，但是机缘巧合，我的个人项目完成的时候，刚好从一个前端论坛看到，腾讯招聘网页重构职位（职位现已更名为 UI 工程师）的应届生，里面的职责和要求跟我的能力很匹配。所以我就加了招聘者的 QQ，给他投递了自己的作品链接。第二天，我竟然被告知，虽然腾讯没有准备在西安招聘网页重构，但是设计中心的总监正好在西安招设计师，可以顺便面试我。

我很珍惜这个面试机会，下决心要好好把握。我首先整理了一页纸就能说明白的个人简历，保持简洁，不重要的信息都删掉这很重要。然后我考虑到，因为我的主要亮点在于我的在线作品项目，如果去面试的时候，面试官没有网络或者电脑看我的作品，那么我辛苦获得的面试机会就白费了。所以，我耗费巨资（1 块钱打印一张）打印出了自己的彩色作品，每一个设计截图打印一张图，放在文件夹中，后来证实派上了很大的用场。

第二天，我在酒店里见到面试官，他是一个温和友善的人，我心里的戒备放下了不少。他问我是怎么获得这个面试机会的，我如实回答。然后我就拿简历和作品给他看，他看了几页作品之后，问我是谁设计的，我说是我设计的，然后我就开始讲我对设计的理解。

这些作品都是我先在白纸上手绘——主要是彩色铅笔——然后用照相机拍下来，再放到 PS 里处理一下对比度什么的。此外，我还加入了一些自己喜欢的元素，然后考虑了一些设计的原则，比如色彩、信息的对齐、一致性和对比等。技术上做了一些 JavaScript 动画，还有 AJAX 交互，通过 cookie 存储信息，等等。

我的毕业求职作品，自己一个人实现了设计和编码。

其实我的设计知识仅限于自学，来自于一本书——《写给大家看的设计书》。这本书非常入门，但是浅显易懂，既有设计理念，也有实际操作，到现在为止我反复看了 3 遍以上。

我理解了书里说的设计四大原则：对齐、对比、距离和重复。虽然我基本没有设计经验，只会一些基本的 Photoshop 操作，但我理解了这几个原则，每次看到好的设计和差的设计时，都能有所感悟。如果不理解，可能我只能用“上流”“高端”“简约”这样空泛的词汇来描述设计。关于设计原则，我在后面的章节中会单独提到。

面试官对设计相关的话题竟然很有兴趣，所以我们在这里聊了很多。后来我才知道面试官是设计总监，所以我提到的一些对设计的基本理解也算是班门弄斧。虽然我应聘的职位是网页重构——一个写代码的职位——不过对设计的认识应该还是会加分不少吧。

后来的事情很简单，我通过了 HR 的面试，就签了。这就是我的故事：投递的第一份简历，也是唯一一份简历，经历的唯一一次面试，和第一份工作，直到现在。

有人会说，大公司的录取通知书很难获得，那都是发给学霸的。其实不是这样的，下面是我的理由。

校园招聘是很多大公司很喜欢的一个人才渠道，因为比起社会招聘的应聘者，毕业生更加有空杯心态、更正能量、更有激情，虽然缺少经验，但是经过一两年的培训也能很快成为团队骨干。而如果是本身有项目经验的毕业生，或者是在 GitHub 上有知名作品、知名博客、去过其他大公司实习的毕业生，那就更加抢手了。至于大学考试成绩，影响不大。

社会招聘的目标是有经验者，招聘时间没有校园招聘那么固定，随时都可能有职位空缺，但是每次放出的名额不会很多。而且这时候会根据招聘岗位，有针对性地考核应聘者的专业能力与综合能力，导致社招的竞争是非常激烈的。

相对而言，我认为校园招聘的门槛并不高，重要的是找对方法。如果您的学校不是顶级，您的成绩不是学霸，那就要走不寻常的道路。

获得面试机会

我的个人简历项目，从构思、设计到实现，前前后后花了一个月时间。现在回头看，这个项目是一个很稚嫩的学生作品，有很多值得改进的地方。但是这个故事的重点是，绕开每个人都会去做的一件事——投简历——转而做一个自己的作品，这就给企业传达了一个信号。

其实很多学生都不知道如何发出自己的信号，只能去做最辛苦的事：去专攻那些性价比很差的信号。

为什么用人单位需要信号？因为无论是 HR 还是需要用人部门，要从茫茫人海中找到需要的人，都需要找一些明显的信号。比如 985 和 211 名牌大学这样的教育背景，就是一个不错的信号，它代表了您的学习能力。虽然大学里学习的专业和知识不一定是企业真正需要的，但是企业需要的是您的学习能力。一个名牌大学毕业生，很有可能有不错的学习能力，假如您能在工作中也保持这种学习能力，那就满足了企业的要求。

但是名牌大学只是第一个信号，经过再一次筛选，HR 过滤掉一些学习成绩差或者沟通能力差的人，您就可以进入到专业面试了。

有些人问，想进大企业是不是一定要高学历？其实不是的，对于 HR 在大样本中的初步过滤，高学历是必需的。但是如果您能够跳过 HR 这一层，直接把信号发送到企业主管那里去，高学历就不是那么刚性的需求了。

所以，无论您是名牌大学的高材生，还是自学成才的专科生，在制作第一份简历的时候，我有这样几个建议。

- 首先确定自己的求职意向，针对特定意向填写您的简历。
- 如果您想表达出自己的创意，不要使用各大招聘网站提供的简历模版。
- 把简历发送到真正在招人的企业主管那里。

举一个例子，作为程序员和设计师，作品是排名最高的信号。在著名开源项目中贡献代码，说明您有能力阅读和编写好的代码，这是公司直接需要的技能。此外，这还能说明您有能力与他人协作：开源代码总是需要协作的。开源项目还能表明您对新鲜事物有热情，表明您也许英语能力不错，有查阅文档的能力……一个开源项目需要的精力也许不会特别多，但它的加分点可就非常多了，简直是一箭 N 雕！

个 人 简 历

填表日期：

姓名		性别		年龄		[贴照片]
地址						
	邮政编码		电子邮件			
	电话		传真			
毕业学校及专业						
应聘职务						
教育						
奖励						
语言						
工作经历						
推荐						
技能						
获得证书						

“不要使用通用的简历模板，图片来自网络。

这样的信号还有很多，但是我不会一个个列出来，每个人都可以自己思考

如何发出自己的信号，毕竟如果大家都发出一样的信号，用人单位就又需要别的信号去分辨人才了。

有些人觉得自己学校不好就没机会了，其实不是这样的，我们的同事有很多就是专科毕业的，如果能力非常强的话，会不止释放出学历这一个信号的。

为什么要把简历发送到真正招人的企业主管那里？因为 HR 没有能力辨别技术能力的高低，他只能根据学历、分数等硬指标来筛选。所以一些技术能力优秀但是分数不高的同学可能就很遗憾地失去了面试机会。

最后的忠告是：基本上大型公司只有在每年的校园招聘期才会招聘学生，平时是没有毕业生名额的。所以，学生还是要多关注校园招聘的时间，在非校园招聘的时间投递简历的话，很容易石沉大海。

实习

一般来说，为了让学生去各大企业实习，学校在大四不会安排太多课程。

实习经历对毕业招聘很有帮助。我的一个朋友，在大三的时候到阿里巴巴实习了半年时间，在大四刚开始的校园招聘中，就拿到了腾讯和百度的录取通知书，而且薪酬比没有实习经历的人还高一点，这就是实习的好处。

实习能提升自己的实践能力，可以认为是从学生到社会人士的一个身份过渡。有的公司在校招的时候，会同时发放录取通知书和实习通知书。因为公司已经认可你的能力，所以实习不是必需的。如果学校没有特别重要紧急的事情，我建议多花点时间去公司实习，熟悉同事和技术流程，并学习项目产品，这对毕业生很有帮助。

以腾讯为例，公司每年在毕业季都会招聘几千个毕业生，然后在实习期间将一大批毕业生放在一个虚拟班，进行一个多月的封闭培训。培训内容包括公司产品的介绍、公司价值观的灌输（小伙伴们亲切地称之为“洗脑”）。

最重要的是，这期间可以认识很多很多的小伙伴。我在实习封闭培训上认识的很多同期生，到现在都是最好的朋友。

培训结束之后，新人们就会回到各自的工作岗位去实习。一般来说，团队领导分配给新人的项目都会比较轻松，一周也就一两个需求的样子，留足时间让新人认识项目中的所有人，熟悉项目中的流程和工具。但是这并不表示新人就可以懈怠，公司有权在 3 个月的试用期内辞退员工。

希望您能有一个好的开始，下面是我的建议。

- **记住团队里的每一个人**

横向的团队是跟您一个专业方向的同事们，纵向的团队是您们作为上下游，为同一个项目服务的同事们。在创业公司，可能没有区分得这么细，都是为产品服务。在大公司，就会有两个维度的同事关系了。无论如何，尽快记住他们的名字和长相。

- **有任何问题，主动问导师**

在腾讯，对于每一个入职的新人，都会设置一个导师（tutor），来帮助新人尽快适应工作和生活。如果是毕业生，一般会设置一些有经验的老员工来做导师，如果是社会招聘的新人，就会设置高级别的老员工来做导师。如果您所在的公司没有为您设置导师，也没有关系，直接去问您的直属领导，让他烦到不行，或者让他告诉您，以后有事情可以问某个人。

- **主动介绍自己，告诉大家自己是新人，请多关照**

这跟新手司机在车后面贴上“新手上路”是一个道理，人们会对新手的天真或者愚蠢有更高的容忍度。

- **每周发邮件记录心得总结、经验教训、学习成长**

总结邮件是一个很好的工具，既可以促使自己反省本周的工作，也可以让领导和其他团队成员了解自己。邮件也不一定要很死板，可以加入一些个

人特色，这能让更多人对您印象深刻。其实不光是新人了，任何阶段的人都可以用邮件记录这种工具来帮助自己。几年前我们部门还没有强制规定新人一定要做每周总结，那时候我已经开始主动记录本周的工作和生活，然后发表在博客上，持续了三十多周。现在我们已经把新人每周邮件总结加入到必需的流程中，并且导师和领导都要回复。

- **实习结束时，用邮件总结所有项目，给出交接文档，并向大家致谢**

如果一些项目在自己离开之后，需要其他人帮忙跟进，这时候交接是必需的。自己主动用邮件总结出来，也会让大家看到您的责任心和专业精神。我在实习结束提交了离职流程之后，才发现机器账号已经没法登入内部网络，而且已经天黑下班，最后只好把交接事项和感谢的话写在团队小黑板上，第二天让导师帮忙拍照发邮件。

第四周流水记 [周记]
08月 15日

Tumblr是神马 [产品]
08月 14日

W3C Unicorn Validator 书签 [做需求]
08月 13日

第三周流水记（王道） [周记]
08月 09日

不要给用户弹出窗口 [做需求]
08月 08日

如果非要说发展方向的话…… [做需求]
08月 04日

第二周流水记 [周记]
08月 01日

第一周流水记 [周记]
07月 25日

开始工作了 [周记]
07月 18日

再见大学 [周记]
07月 05日

实习期间我在博客上写的周记。

说了这么多，好像跟全栈工程师没什么关系，但是要知道，**主动性是全栈工程师必须具备的一个特质，没有一个全栈工程师是被别人逼出来的。**

延伸阅读

1.《编程之美：微软技术面试心得》《编程之美》小组，电子工业出版社

野生程序员的故事

野生程序员是指仅凭对计算机开发的兴趣进入这个行业，从前端到后台一手包揽，但各方面能力都不精通的人。野生程序员有很强大的单兵作战能力，但是在编入“正规军”之后，可能会不适应新的做事方法。

遭遇“野生程序员”

腾讯公司内部的团队很多，在团队管理上有项目和专业两个维度。也就是说，有些团队是项目维度的，整个团队共同维护一个产品，成员来自不同的职业岗位；有些团队是专业维度的，比如一个组都是前端工程师，维护不同的产品。

因为前端组是设计部最接近后台技术的团队，所以团队平时的工作和技术交流分享，都不局限于前端技术领域，还包括很多服务器端或者移动端的技术。从前端到后端，一些技术问题都要我们自己来解决。

在招聘前端工程师的时候，我们对应聘者的要求是，在掌握基本前端技术的前提下，最好有更为全面的技术。这样，即使我们的项目人力结构、平台和方向发生变化的时候，他也能够更加灵活地转移到其他角色中。而且技术的全面更能表现一个人对技术的热情以及较强的学习能力。从团队多样性来讲，多一些技术种类的话，大家在一起也能碰撞出新的火花。

有一次，我在 QQ 群发布了一条简单的信息：“招聘前端工程师，全栈更佳。”随后有一个“全栈工程师”A 君向我自荐。

我仔细看了他的简历：“三年工作经验，擅长 PHP、MySQL 数据库、jQuery、HTML 和 CSS，对 CDN 加速和网络安全也颇有研究。”他的简历让我眼前一亮，于是我跟他进行了一次简单的电话面试。

电话面试的第一个环节照例是让 A 君简短地介绍自己。A 君在一个传统行业的小公司做 IT 技术支持工作，公司的 3 个网站项目都是他一手搭建，从架构到编码细节他都如数家珍。他号称能解决一切技术问题，老板提出的所有需求都能完成，而且只有他能完成。随着最近公司业务量越来越大，他还招了两个下属，但是主要的编程工作还是他在做。

我问他：“我们的职位是前端工程师，那么您有哪些前端方面的技能呢？”他回答：“我擅长 HTML、CSS 和 JavaScript。”

“对于 Web 性能优化，您有哪些了解和经验吗？”他思索了一阵答道：“我们在发布项目之前压缩 CSS 和 JavaScript 源代码，这样文件体积就变小了，用户加载必要资源所花的时间也就更短了。”我继续说道，很好，还有吗？他想了半天，答不上来了。

其实关于 Web 性能优化，有非常多的方面可以去做，我希望应聘者能尽量多回答一些。

- 压缩源码和图片

JavaScript 文件源代码可以采用混淆压缩的方式，CSS 文件源代码进行普通压缩，JPG 图片可以根据具体质量来压缩为 50% 到 70%，PNG 可以使用一些开源压缩软件来压缩，比如 24 色变成 8 色、去掉一些 PNG 格式信息等。

- 选择合适的图片格式

如果图片颜色数较多就使用 JPG 格式，如果图片颜色数较少就使用 PNG 格式，如果能够通过服务器端判断浏览器支持 WebP，那么就使用 WebP 格式和 SVG 格式。

- 合并静态资源

包括 CSS、JavaScript 和小图片，减少 HTTP 请求。

- 开启服务器端的 Gzip 压缩

这对文本资源非常有效，对图片资源则没那么大的压缩比率。

- 使用 CDN

或者一些公开库使用第三方提供的静态资源地址（比如 jQuery、normalize.css）。一方面增加并发下载量，另一方面能够和其他网站共享缓存。

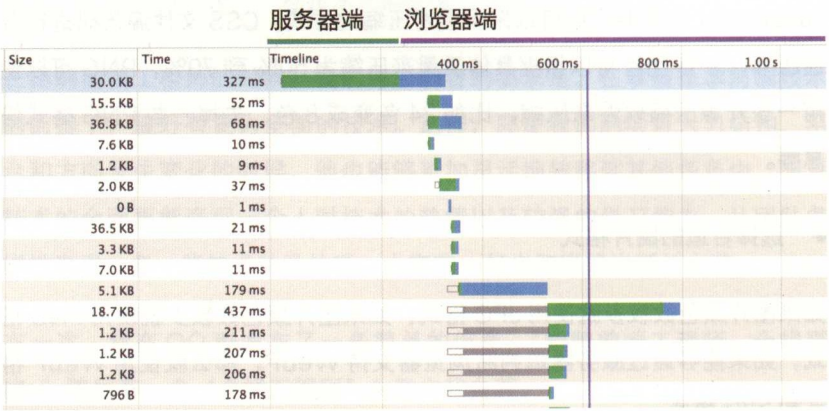
- 延长静态资源缓存时间

这样，频繁访问网站的访客就能够更快地访问。不过，这里要通过修改文件名的方式，确保在资源更新的时候，用户会拉取到最新的内容。

- 把 CSS 放在页面头部，把 JavaScript 放在页面底部

这样就不会阻塞页面渲染，让页面出现长时间的空白。

每一个条目都可以进一步深层挖掘下去。



Web 性能优化分为服务器端和浏览器端两个方面。

此外，由于中文的歧义性，Web 性能优化这个词既可以解读成页面加载速度（Page Speed）的优化，也可以解读成页面渲染性能（Page Performance）的优化。或者是二者的集合。所以，应聘者如果能在这个问题上多做一些分析，会有很高的加分。但是 A 君在网络性能方面的研究只是浅尝辄止，停留在压缩资源方面，这说明他还没有足够理解 HTTP 协议本身。

关于网络性能和 HTTP 协议，作为大公司的前端工程师是非常看重的，因为每一个页面都会有亿万用户访问量，任何一点对服务器带宽压力都会积少成多，最终造成很大的成本。关于这方面的技术详解，我在后面会有一篇单独的文章来分析。

接着上面的故事，我想既然他对 Web 性能优化方面不太熟悉，可能他是一个偏后台的程序员，因而就又问道：“关于服务器端 MVC 架构的技术实现，您是怎样理解的？”他说：“是数据模型、视图、控制器的分离。”

我更进一步问道：“这种架构方式有什么好处？您在项目中是如何应用这一架构的？”他回答说：“MVC 的架构方式会让项目可维护性更高，所有涉及界面的代码都在视图（View）里面，所有涉及核心逻辑的代码都在模型（Model）里面，URL 路由之类的代码都在控制器（Controller）里面。我在项目中使用了 MVC 架构的 PHP 框架——CodeIgniter。”

我一边打开他的网站，一边继续跟他电话沟通。当看到网站的 CSS 代码都直接内嵌在 HTML 头部的时候，我忍不住问他：“为什么您的网站的 CSS 代码都内嵌在 HTML 里面呢，是使用自动化工具合并进去的吗？”他支支吾吾地说：“因为在本地调试的时候，CSS 文件修改经常不生效，所以就直接在 HTML 里面改了，这样比较快。”

好吧，**我想这是一个典型的“知易行难”的开发者**，他知道采用 MVC 架构的项目的可维护性更高，可是在分离样式与结构上面还没有达到最基本的要求，甚至把 CSS 写在 HTML 中。至于他说的在本地环境上发现 CSS 文件经常缓存，可能要看看本地服务器的缓存设置是否有问题，然后再做调试。稍微了解一点 HTTP 的浏览器端缓存，这就不是难事了。我更欣赏在开发流程上花工夫去理解和优化的应聘者，而不是马马虎虎，只是以完成需求为目标的人。

我突然想到他说的“所有需求他都能完成，且只有他能完成”，于是就想问问他代码版本管理方面的问题。我说：“您们团队现在加入了两个新人，那么您们如何进行代码版本管理？”他回答：“我们有一台测试服务器，用 FTP 来测试代码，如果在测试机上没有问题的话，我们就会发布到生产环境。”

我说：“等等，我不是问您们代码部署的问题，是平时您们如何管理代码版本，如何分工协作的？”他说：“我们把代码从测试服务器上拷下来，修改完了之后再传上去。”

到这里，我终于明白为什么他们团队的新人无法快速融入项目了，因为项目没有使用 SVN 或者 Git 这样的版本管理工具。团队只有一个人在写代码的时候，缺乏版本管理工具的问题可能还不会暴露出来，但是当更多成员加入时，整个项目就会寸步难行，大家都要花大量的时间合并代码，以及找回丢失的代码。万一出现了外网 bug，版本工具也能帮我们把站点状态快速恢复到之前的时间点。在本书的后面章节，我会详细介绍版本管理工具。

最后我抱着几乎绝望的心情，问了下关系数据库设计原则方面的问题，他的回答也不是很理想。

我知道，我又遭遇了“野生程序员”。

什么是“野生程序员”

所谓“野生程序员”，就是没有计算机基础知识和相关教育经历，靠着对计算机开发的兴趣进入这个行业，虽然知识面比较广，但是各方面都一知半解的开发者。

这几年我从一个求职者，转变成一个招聘者，有一个感受就是，中国高等教育与市场需求不接轨。学校不了解市场究竟需要什么样的人才，其设立的课程和技术往往比市场技术现状落后了 5 年以上。我在大学学习用 ASP 建站，但是现在已经几乎没有人用 ASP 建站了。一个直接的后果是，很多高校毕业生不能满足企业的要求。

与此同时，中国互联网市场蓬勃发展，特别是移动互联网的发力，让中国跳过“WAP 时代”，直接进入“App 时代”。市场的热钱都投入到互联网行业，“BAT”等大公司不断扩张，创业公司也如雨后春笋，整个市场对软件工程师的需求缺口巨大，所以很多公司在招人的时候，没法招聘到“专业”的计算机专业毕业生。

在美国，因为教育与市场稳定发展了很多年，供求关系相对平衡，计算机相关专业本科已经成为基本要求。举例而言，美国的硅谷公司（如 Google）绝大

部分前端开发招聘岗位都有一个最低要求——本科学历，计算机相关专业。

相比而言，从中国的大公司（如腾讯）的招聘网站上可以看出，有一些前端开发岗位没有对学历的要求，也有一些要求“本科及以上学历”，少数才会要求“本科学历，计算机相关专业”。我们的团队中就有一些成员是大专学历。许多企业在招聘的时候往往放松了对学历的要求，只看重项目和经验，而不看重学历。这是一件好事，代表市场在高等教育的规模和质量都跟不上市场要求的情况下，给予更多有兴趣和能力的年轻人进入 IT 领域的机会，也填补了人才市场的空缺。

美国硅谷，是世界互联网公司的中心，是所有求职者梦寐以求的圣地。在最开始，硅谷之所以名字当中有一个“硅”字，是因为当地企业多数是从事加工制造高浓度硅的半导体行业和电脑工业。随后，互联网公司和软件公司渐渐取代传统的硬件公司，让硅谷获得了新的生命，但硅谷这个名字保留了下来。在硅谷从诞生到发展壮大的整个生命周期中，斯坦福大学起到了很大的作用，我认为称之为硅谷的母亲也不为过。

在中国，由于政策、环境、历史原因，还有大学教育投入上的差异，导致大学在整个互联网发展中起的作用没那么大。中美两国 IT 人才市场供求关系上的这些差别，也反映在整个行业文化中。

一个直观的反映就是软件工程师的“草根”化。其实很多软件工程师的收入都很高，处于中上层水平，相比金融行业的白领也毫不逊色，但是一谈起程序员，大家的印象还是“一年四季的 T 恤（在行业展会上免费拿的）牛仔裤，平时也喜欢宅在家里，不会像同样收入的金融白领，平时爱好听歌剧打高尔夫球”。这种差异一方面是外部人士对软件工程师职业的偏见，另一方面也是程序员行业的自黑习惯。在招聘时岗位要求就已经放到最低：不要求学历、上班不要求着装、上下班时间灵活，这样才好更方便地招聘。而金融行业有意识地塑造一种“精英”文化，从学历就设置高门槛，即使有些工作根本不需要那么高的学历。

回到毕业生的话题，很多跨专业的学生发现自己兴趣在互联网和计算机方

向的时候，就开始了自学之路，基本上学习方式有这样几种。

- **书**：在计算机图书领域，技术难度跟图书销量是成反比的，从标签教起的 HTML/CSS 基础书籍卖得最好，其次是关于 JavaScript 和 jQuery 的书，Angular 和 Node.js 之类的就没那么畅销了。

- **互联网**：得益于全世界都在互联网上共享的资源，现在的学习者有了更多的选择，比如关于 Web 开发基础教学的 W3CSchool，还有海量的技术博客。我个人喜欢订阅一些英文大站，比如 Smashing Magazine（http://www.smashingmagazine.com/）、tuts+（http://tutsplus.com/）等。我在读大学的时候，Google Reader 还没有永久关闭，那时候我很喜欢用 RSS 来关注这些站点的更新情况。Google Reader 下线后，就基本上废弃了 RSS 阅读的习惯，转而用一些社交网站来追踪更新情况，但是有时还是会淹没在大量无用的信息里面。

- **社团**：学校的网站社团也孕育了许多能力很强的开发者，社团经过历届的传帮带，技术有所积累，比如师兄会教师弟用 Sublime 编辑器，这就比还在用 Dreamweaver 的同学更有优势。此外，学校社团有一些定点客户，比如学校教务处、周边商户，所以有更多的实战经验，在毕业时作品集也丰富了不少。

因为有这样一些自学渠道，所以不一定只有计算机专业毕业的学生才有机会进入互联网行业。毕业之后，这些计算机爱好者进入不同的工作岗位，不同的是，有些进入大公司，有些进入小公司。这两者的成长轨迹往往会不太一样。

小公司有很多野生程序员

流水线工作流程有诸多优点，但一般来说，大公司才需要很多专精某种技术的工程师，组成一个 Web 开发团队。创业公司只需要几个技术全面的人来做开发和技术支持，有时候甚至只有一两个人而已。

当然，最主要的原因就是成本和回报的问题。招聘和维持庞大的 IT 研发团队需要一笔不小的开支，小公司并没有那么多 Web 服务的需求，一般企业可能只需要一个公司站点就可以了，现在甚至完全不需要 Web 站点，可以用微信公共账号或者淘宝这样的大平台来完成。如果招聘一个完整的 Web 研发团队，从用户研究到交互设计、从 App 开发到数据库管理，直接后果就是整个团队大部分时间都空闲着，无事可做。与之相比，聘请一个或多个全栈工程师会更高效、更省钱。

第二个原因是，很多传统线下公司并不会特别依赖 IT 技术，有些时候线下渠道占据了公司大部分收入来源，所以公司不需要架设十分完善的线上服务。由于线上服务的用户量少，所以 Web 服务对稳定性、承受压力、用户体验的要求都没有那么高。此外，由于没有太多重要的用户数据，所以异地容灾也不需要。

因为公司的开发团队小，所以网站无论出现什么问题，都需要他们去解决。从域名到服务器，从前端到后台，从设计到内容，都是一人包揽。野生程序员了解的知识越来越多，但是样样都不精通。我认识几个小公司的程序员，他们没有明确的职称，开发者都统称为程序员，设计师都统称为美工。

在 Web 技术的任何方向，比如前端开发或者服务器端开发，他们既没有很强的经验，也没有明确的兴趣。那么当他想跳槽到大公司的时候，会发现大公司对岗位和职责的细分非常明确，而自己的能力达不到某个细分岗位的要求。所以他们很难在专业上继续进步，从而陷入原地踏步的窘境。

大公司还是创业公司

在许多论坛上，常常会看到毕业生提出这样的问题：现在有一个大公司和一个创业公司的机会摆在我面前，我应该选择哪一个？其实每个人有不同的想法、不同的风险偏好，旁人没办法针对这个宽泛的问题给出标准的答案。但是既然提问者是毕业生，这种情况下我还是建议选择大公司，因为

会选择创业公司的人往往有自己的主见，已经接受创业公司的邀请去工作了，不会去发帖询问大家的意见。当然这是开玩笑，真正的原因是，在大公司的头两年，是从学生到职场人士的一个转变，您可能会从大平台学习到一些规范的流程方法，养成一些足以影响您一生的习惯，认识更多的能对您职场有帮助的人脉。

大公司能给您的

- 较小的风险

每个公司都有倒闭的可能，但是，显然大公司比小公司的风险低多了。如果您的风险承受能力较低，那么不得不考虑这个因素。

- 技术最佳实践

在大公司，对代码质量和一致性的要求很高，所以一般在最终发布前会有代码审查（Code Review）流程和项目总结会等。如果您完成了一个任务，但是没有采用最佳实践，只是 hack[1] 了一下，那么其他同事可能都会指出您的问题，并且要求您改正之后再提交。小公司或者创业公司人力比较紧张，在他们看来，快速实现和上线，比优雅地上线更重要，所以对于一些最佳实践类的问题，只能睁一只眼闭一只眼啦。

- 垂直专精的技能

大公司专业分工很细，而且有更多技术沟通和沉淀的氛围，所以容易让人在垂直专精的技术方向有足够的发展。在小公司更能锻炼技术的广度，深度上缺乏锻炼的环境。但是其实二者的利弊，都是外界的，技术人员的个人成长除了工作时间的锻炼，还要靠下班后的时间，外界只是给予一个环境或者机会。

---

1 所谓 hack，就是不优雅的解决方案。比如一个界面的调整，如果采用最佳实践，需要用 MVC 架构来分离出界面相关的代码，并且把有可能相关的变量提取出来，合理命名并且放在合理的位置。如果是 hack，可能就不管这么多，看见哪里需要修改就原地修改了，表面上看很快解决了问题，可是这会给后面跟进的同事造成很大的困扰。

- 服务海量用户的经验

同样是做一个网站，服务少数用户量和服务海量用户量时需要考虑的事情是完全不同的。小网站遇到的问题，大网站一定遇到过，而大网站遇到的问题，小网站就不一定遇到过了。当一个网站发展到业内最强时，它的问题没有人遇到过，这时候就不能凡事问百度、Google或Stack Overflow了，而要自己去探索解决方案。

- 软技能

硬技能是指每个职位需要的专业技能，软技能则是通用的技能，比如沟通、影响力、项目管理和演讲等。越是大公司，越是看重影响力，所以会有很多培训教您如何提高影响力。

我在面试一些来自小公司的应聘者时，就发现他平时的工作中，周边环境很少有分享和沉淀的习惯。沉淀和总结是很重要的，在腾讯，设计师做完一次设计定稿之后，就会把设计的思路，包括整体的设计风格、设计规范和色彩的确定等都总结成一封邮件或者PPT，发送给部门同事。每个人都要有意识地维护自己的作品集，它在半年一次的考核、晋升面试甚至以后的跳槽中都非常有用。但是小公司的设计师不太会总结个人作品集，时间紧急是一方面原因，另一个主要原因是环境不需要他这样做，因此就缺乏了这方面的锻炼。

- 人脉

每年都有不少人从大公司离职去创业，这是非常自然的事情。对于大公司出来的人来说，之前积累的人脉资源这时候会起到很大的作用，比如创业期间的一些合作机会或者资源的互利，等等。万一创业失败，也不会很惨，因为您之前接触的人脉可以给您提供工作机会。但如果您刚毕业就选择创业，创业失败之后没有人能给您提供工作机会。

- 心态

其实大公司能给予毕业生最大的优势，就是提供一个心智培育的土壤。之

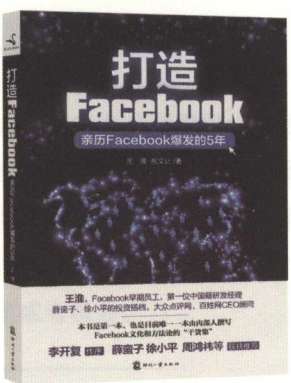
前参加面试官培训的时候，我大概了解过公司招聘一个毕业生投入的成本。从校园招聘，到安排面试官面试候选人，再到封闭培训和一些课程培训，再给一段时间熟悉项目，最后 3 个月试用期后可能还要淘汰掉一些。如果把成本平摊到每一个人身上，这些投入要一年才能收回来。而小公司不会有这么大的耐心去培育一个新人。如果没有足够的时间去学习和成长，可能在一两年后，员工的能力也比较全面，但是样样都不精通，也说不清楚自己的目标是什么，于是就变成了“野生程序员”。

综合来讲，在大公司中，从硬技能到软技能都会有很多经验丰富的前辈能够教您，您会在大平台上学习到很多东西。工作几年之后，员工的选择也很多，要么走技术路线继续发展下去，做高级工程师；要么学习管理和领导力；要么出去创业。

所以，我的个人建议是，**从毕业生自己前途发展的角度来看，先加入一家上市大公司是个不错的选择**。

延伸阅读

1.《打造 Facebook》王淮，印刷工业出版社



工程师事业指南

我曾读过一本有意思的书,《您就是极客》，副标题是“软件开发人员生存指南”。其中第二章专门讲软件工程师事业的 3 个关键词，分别是技术、成长和声望。前面的文章里已经讲了技术和成长，现在我们来谈谈声望。

那个什么都懂的家伙

积累作品集

突出重点

那个什么都懂的家伙

让我先描述一个场景：作为一个软件工程师，您现在遇到点棘手的问题，项目不知道怎么进展下去了，这时候您会去找谁？这时候您的脑海中应该会浮现一个家伙的名字，这个家伙可能穿着展会上送的免费 T 恤和拖鞋，也不太爱参加集体活动，但这不重要，重要的是他能稳定地解决问题——就像机器一样。

没有什么他做不了的，没有什么他解释不了的，而且您永远也争论不过他。

既然他这么讨厌，那么为什么您遇到麻烦的时候还要找他帮忙呢？答案非常简单，**因为他什么都能解决**。您不用思考就知道他一定能帮上忙。即使您的问题领域是他不熟悉的，您也知道他一定能帮您。

表面上看，大家是因为他的技术全面而找他，实际上并不是，真正的原因是声望。怎么表达这种感觉呢，就好像您跟上司说“这个问题是很棘手，但小明能帮我解决”的时候，上司脸上宽慰而庆幸的表情。

是的，就是这种“无论多么难搞，他肯定能解决”的信赖，把他跟其他人完全分开，当下一次考评或者升职的时候，上司第一个会想到谁呢？

没错，就这么简单，“声望”就是这种日积月累的印象。软件工程师事业指南告诉您，最核心的 3 个词就是技术、成长和声望。技术是您的武器，成长就是好好打磨武器，而声望是您一生的积累。

怎样获得良好的声望？很简单——答应做的事，全部都要完成。

说起来简单，操作起来很困难，让我们来做个数学题。

- 每天您会接收到一些请求，有大有小，我们假设数量是 X。
- 您每天能完成或者计划能够完成的请求，我们假设数量是 Y。

- 如果 X 大于 Y，您的声望就受损了。而其中一些请求对声望的影响会比其他声望更大。

您会觉得偶尔完成不了一个请求没什么大不了的，别人不会在意的。事实上，他们在意。也许只有一瞬间，但在那一瞬间，他们会认为您没有跟进，没有完成，没有上心，这也是您最终留给大家的印象。

那如果上司真的给出一个非常棘手的问题，您该如何回答？没错，您不能直接拒绝。拒绝上司是很困难的。但您也不能什么都答应下来，随后又无法完成任务。那时候您会丢掉更多的得分。

正确的方法是，讲出事实。

比如您希望老板加派人手的时候要说："我现在手头已经有某某工作，现在给我这个新的任务，时间会来不及。看看能否把之前的工作移交给小明，或者让小明跟我一起完成这个新的任务。"

或者您也不知道怎么做的时候，可以真诚地说："我很在乎这个问题，也很想解决它，但我现在还不知道怎样才能完成它，帮帮我。"

虽然很有可能最终老板不会给您加派人手，但他会帮您权衡优先级，并跟项目负责人去沟通项目目标。更重要的是，他知道您在努力了。

积累作品集

作品集（portfolio），是指您个人的项目和作品的集合，一份精心准备的作品集比简历更能说服人。

看到这里，您可能会说，这就不是简历中的一栏——项目经验吗？为什么要用这么高冷的名字。这是因为传统程序员的个人简历中过于忽略作品集的包装和展示，所以我想扭转您对于作品集的观点。

重视作品集

在讲如何制作作品集之前——以及做任何事情之前——有一个问题必须要回答：**我做这件事情的目标用户是谁？**

不可否认，程序员制作和维护自己的作品集的最大动机便是更好地向其他人展示自己的能力。那么是哪些人呢？

- **老板：**如果您每隔一年就要经过考评和自证您有多优秀，那么在平时维护作品集就是非常有用和高效的。
- **潜在客户：**如果您是 iOS 开发的自由职业者，那么列出您曾经完成的项目会对您下一个潜在客户非常有吸引力。
- **潜在雇主：**如果您准备跳槽，或者寻找一份新工作，那么潜在的雇主或者 HR 会很在意您的作品。潜在雇主可能是技术领导，也可能是老板。
- **潜在朋友：**其他技术和志向相似的人可能会通过您的作品集找到您，寻求合作项目，或者仅仅是聊聊天。
- **任何人：**对，告诉他们我就是这么牛。
- **自己：**因为自己记性不太好，所以作品集就是一个历史项目的沉淀。

传统的作品集，图片来自网络。

我很重视作品集，一方面体现在我很在意维护自己的作品集，另一方面我也很喜欢面试的时候看到应聘者有自己的作品集。除了工作上安排的项目，我更在意一些课外项目，因为它展示了您的兴趣和热情所在。对于我这种重视作品集的态度和做法，有些人会表示反对。这种反对基本分为两个原因，我来分别谈谈。

第一种批评者是典型的理想主义者，他们认为重视作品集这件事太功利：强调作品集在职场和工作中的重要作用，会让编程这件事失去它本身的纯粹性。程序员编程是为了改变世界，或者享受自己的乐趣，其他的好处是随之而来的，而不应该为了丰满自己的作品集而去做项目。

我的观点是，从某种程度上来讲，重视展示项目这种态度确实会对编程的纯粹性有所腐蚀（如果您编程本身只是为了自己的兴趣），您编写一个项目的动机可能会从纯粹为了好玩，变成获取收益。但是在这个商业化的市场里，对方（高效地）得到了您的信息，您得到了您应有的评价，这对双方是互利的。

为编程这件事引入了任何激励都会影响它的纯粹性，这一点无法避免。但是程序员酬劳的市场化会为雇主提供更好的服务，为程序员提供更好的生活，这就够了。

第二种批评是，制作和维护作品集这件事是很花费时间的，而这些时间本可以做一些更有用的事。

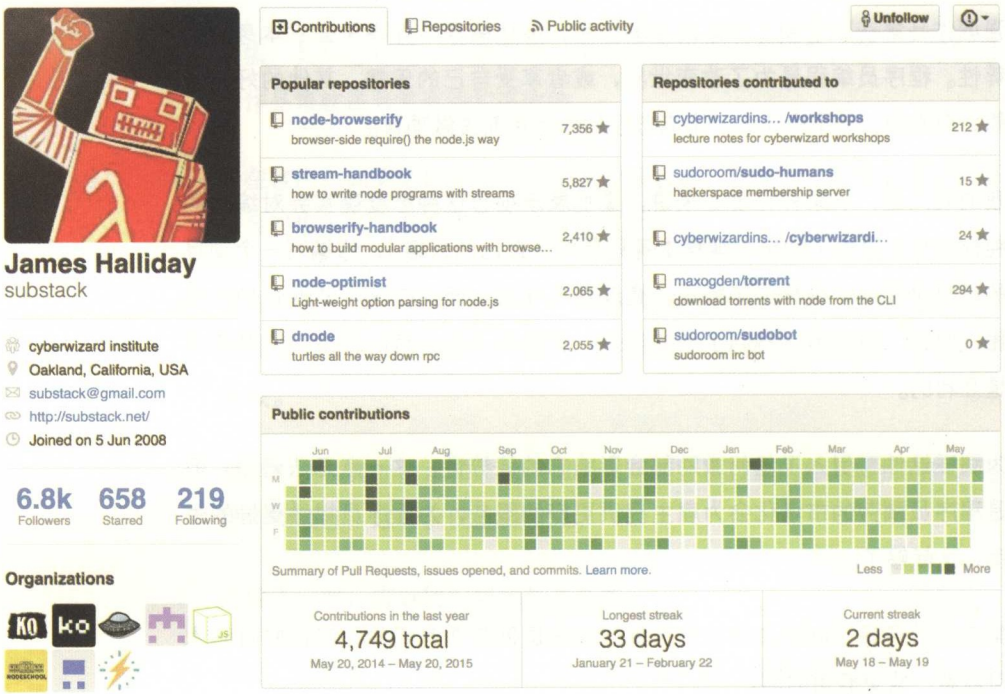
我的回答是，制作作品集本身也可以学到很多，接下来我会讲讲如何用一些很酷的方式制作您的作品集。有些作品集，比如 GitHub 本身，不需要花费额外的成本，反而可以帮助您优化工作流程，或者让您坚持编程。

工程师的作品集

传统的作品集往往是一个纸质文件袋，里面放着一些打印的过往作品。而

在网络时代，印刷品本身既浪费资源也不高效，所以程序员和设计师的作品集往往是自己的在线个人网站。对于程序员来说，成本最低的一种作品展示方式就是把自己的代码发布到 GitHub 上。

因为 GitHub 的开源性和社交性，使得它成为最容易使用的作品集，而它对于懂技术的目标用户群，也是非常直观易用的。



substack 的 GitHub 主页，截图来自 github.com。

这是 Node.js 社区最有名的开发者之一 James Halliday 的 GitHub 主页，他的网名是 substack。从这个主页中可以看到，他有很多非常受欢迎的代码库，而且有很多人关注他的时间线。更重要的是，他几乎每天都在提交代码，这说明他是一个勤奋的开发者。

我曾经有一段时间对淘宝客 API 比较痴迷，希望通过开发一个系统来赚点零花钱。我花了一点课外时间做了淘宝客的导购系统——33 号铺（33pu），

使用的是 CodeIgniter、jQuery、Bootstrap 等一些快速开发框架。

代码方面并没有什么值得夸耀的技术，因为我在 PHP 开发上几乎没有经验（除非改改 WordPress 主题也算 PHP 开发经验），所以代码质量确实比较差。但我根据对用户交互的一些理解，利用各种框架和 API 算是把这个系统架构搭建起来了——我甚至把主要功能做出来之后，才想到登录安全方面的问题。

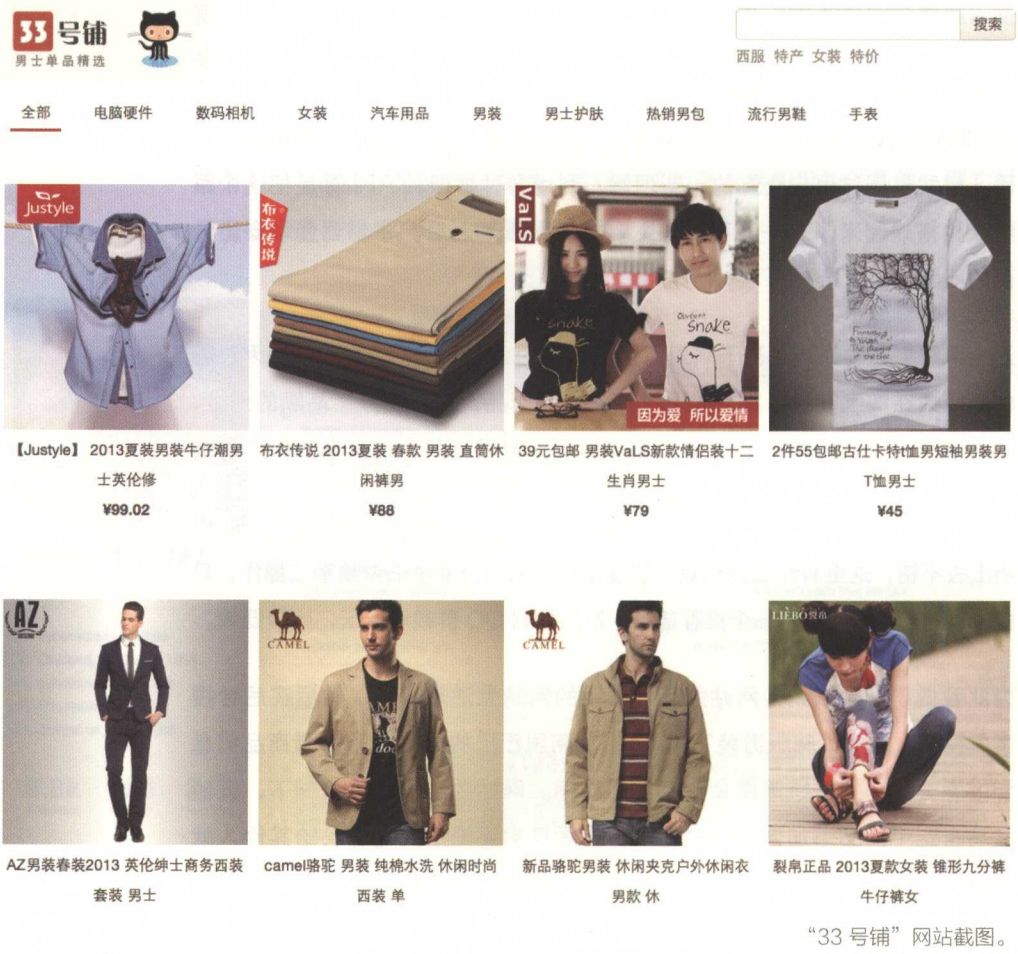
这个系统主要的想法是基于阿里巴巴淘宝客的 API。有一些淘宝店和天猫店的老板会在淘宝客这个官方平台推广自己的商品，同时设置了高额的佣金，比如 30%。那么假如一件衣服价值 100 元，只要任何人向其他人推广卖出了一件衣服，自己就可以得到 30 元钱。

听上去不错，这里有什么问题呢？那就是官方的后台系统很依赖手工操作，只能一件一件推荐，得到一个推荐链接，然后发到 QQ 群或者论坛，比较低效。

为此我做了一个系统，网站主可以在我的网站上注册账号，然后在后台搜索关键词，比如“韩版男装”，系统会从阿里巴巴的 API 拉取大量商品信息和推荐链接，并且按照佣金比例高低排序。网站主只需要点击一下，就能立马将这个商品添加到自己的主页，随后只需要把主页地址发给其他人就可以了，而且对购买者来说，体验更好，因为一个页面上都是聚合推荐的商品。

系统完成之后，我在 V2EX 上宣传了这个网站。有一些朋友对这个系统比较感兴趣，甚至想要花钱买我的系统。我想自己辛苦工作这么久，不能就这么免费开源了，而且我有点忧虑，担心其他人会快速复制我的网站，然后把我打败。

上线一段时间之后，我发现由于缺乏运营，所以流量非常少，后来我看到一篇名为“Open Source (Almoset) Everything”的文章，作者是 GitHub 的联合创始人，我想引用其中一句话：



“33 号铺”网站截图。

> “If you do it right, open sourcing code is great advertising for you and your company.”（如果使用得当，开源代码是您和您的公司最好的广告。）

我想，开源对我也没有什么坏处，他们复制我的网站之后并没有抢走我的客户，**因为我根本没有任何客户，事情不可能更糟了。**相反，如果我的系统足够成功，可能会带来一些流量。而且开源代码如果够出名，会有其他人来帮我共同完善代码。

所以，我把代码放在了 GitHub 上开源，并在 V2EX、腾讯微博、Twitter 等一切我知道的地方宣传了这个消息。当然在那之前我还对代码做了一些优

化，包括把之前硬编码到代码中的配置信息全部挪到 config.php 中，并把（几乎）所有的函数都加上了 phpdoc 注释——对了，因为我是一个 PHP 小白，我还专门查阅了 phpdoc 文档，学习怎样写注释。

我甚至还优化了安装程序，这样便于别人尽可能轻松地部署代码。

代码发布出去之后反响非常热烈，在 V2EX 上有 82 人收藏这个帖子，当天网站流量达到 1200 多 PV，比平时的 30PV 多出了 40 倍，在 GitHub 上两天内有 100 多个 Watch 和 20 多个 Fork 信息，然后我偶然查看 GitHub 的 PHP 语言排行榜的时候发现，当周最热门的项目中，33pu 竟然排名第四！

我的项目进入 GitHub 当周，Forked 和 Watched 位列排行榜前五名，截图来自 github.com。

作为一个 GitHub 新人和 PHP 小白，我觉得这太酷了！

实际上，我还不够了解 PHP 这门语言，但我勇敢地开源了。我觉得这件事挺有意思，最重要的是，自己在这个过程中获得的提升比其他任何人都多得多。

后来其他人发现了程序的一些漏洞，也提交了修复代码，我从中也得到了技术的提高。

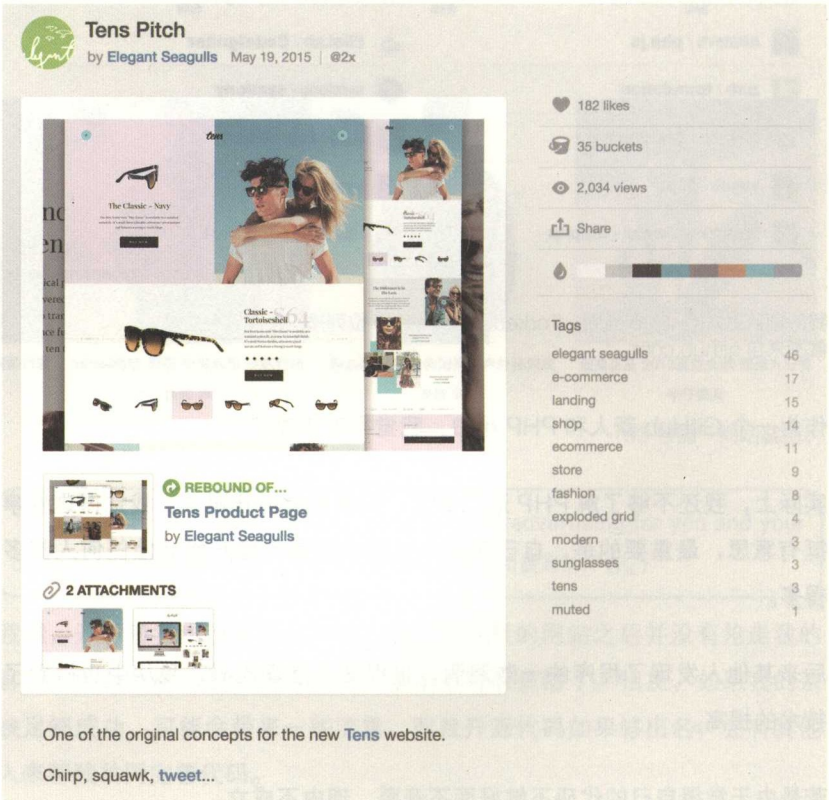
若是由于觉得自己的代码不够好而不开源，理由不成立。

写得不好没关系，《代码整洁之道》（Clean Code）中说，每次签入代码都

保证比签出的时候更好，那么代码仓库会变得越来越好，这必然是一个不断变好的代码仓库。

另外，将代码开源，大家看到的是项目功能，而不是代码技巧。如果不是自己需要，没有人会闲得帮其他人优化代码。如果您的想法够好，那么就会收获来自社区的感谢、帮助，以及您应有的声望。

顺便提一下，如果您是擅长设计和编程的全栈工程师，并且对自己的设计能力非常有自信，那么同样推荐 Dribbble。Dribbble 是设计师的舞台，它的社交性让您的作品很容易传播和收获“赞”。如果是可以实际预览的页面，您可以在贴上设计稿之后，在下面留下站点的实际地址。



Dribbble 可以上传设计图片，是设计师的社交圈，截图来自 dribbble.com。

GitHub/Dribbble

- 格调：☆☆☆☆
- 易用：☆☆
- 成本：☆

我想推荐的第二种方案是静态页（比如 GitHub Pages）。

GitHub Pages 是 GitHub 在代码托管之外额外提供的一个非常方便的功能，它允许您创建一个 gh-pages 的分支（如果是用户或者项目的主页，就是 master 分支），然后向其中提交静态资源，包括 HTML、CSS、JavaScript 和图片，然后就可以通过 username.github.io 来访问。

我的个人博客就是建立在 GitHub Pages 上，因为我的用户名是 yuguo，所以对应的域名是 http://yuguo.github.io/ 。如果您访问的话，会跳转到 http://yuguo.us/，因为 GitHub 提供免费域名绑定功能，这简直是业界良心，所以我绑定了自己的私人域名。

GitHub Pages 的初衷是为您的项目提供一个简单的介绍页，它提供了一些固定的模板。在 GitHub 网页上直接选择这些模板，就会在您的某个项目中创建一个 gh-pages 分支，并且允许您在网页上使用 Markdown 格式直接编辑 index.html 的内容。所以在那个时代，所有的 GitHub Pages 的设计都局限于 GitHub 官方提供的几套默认模板。

后来，Jekyll 改变了游戏规则。Jekyll 是一个使用 Ruby 编写的博客站点编译软件，通过命令行来操作。用户只需要编写 Markdown 格式的内容“源文件”，就能快速编译出一个完整的静态网站。技术的发展总会带来新的应用场景，GitHub Pages 与 Jekyll 结合在一起，发生了美妙的化学反应。现在只需要把 Jekyll 的日志源代码 Markdown 推送到 GitHub Pages 站点，就能生成一个编译后的静态页。

Jekyll 让您可以利用简单的几行代码，就新建一个站点框架，截图来自 Jekyllrb.com。

GitHub Pages 支持 Jekyll 编译之后，用户只需推送源代码到 GitHub，GitHub Pages 就能自动编译。二者产生了奇妙的化学反应，GitHub Pages 的灵活性变得无限大，越来越多的开发者使用 GitHub 托管博客，而作品集也是一种非常适合 Jekyll 生成的项目。

除了 Jekyll 这种博客编译器以外，还有一些专门的静态站点编译器，比如 Dexy。与 Jekyll 不同的是，Dexy 更擅长产品站点和文档的编译，比如可以直接引用某代码文件到 HTML 中。Dexy 不被 GitHub 原生支持，所以您可以在本地编译出完整的静态页面之后，把生成的站点推送到 GitHub Pages。

经常有人问我博客托管在哪个服务器，我会告诉他们托管在 GitHub Pages，虽然速度不是特别快，但是很稳定，可用性可以保证在 99.99% 以上。

GitHub Pages + Jekyll/Dexy

- 格调：☆☆☆☆☆
- 易用：☆☆☆☆
- 成本：☆☆☆

突出重点

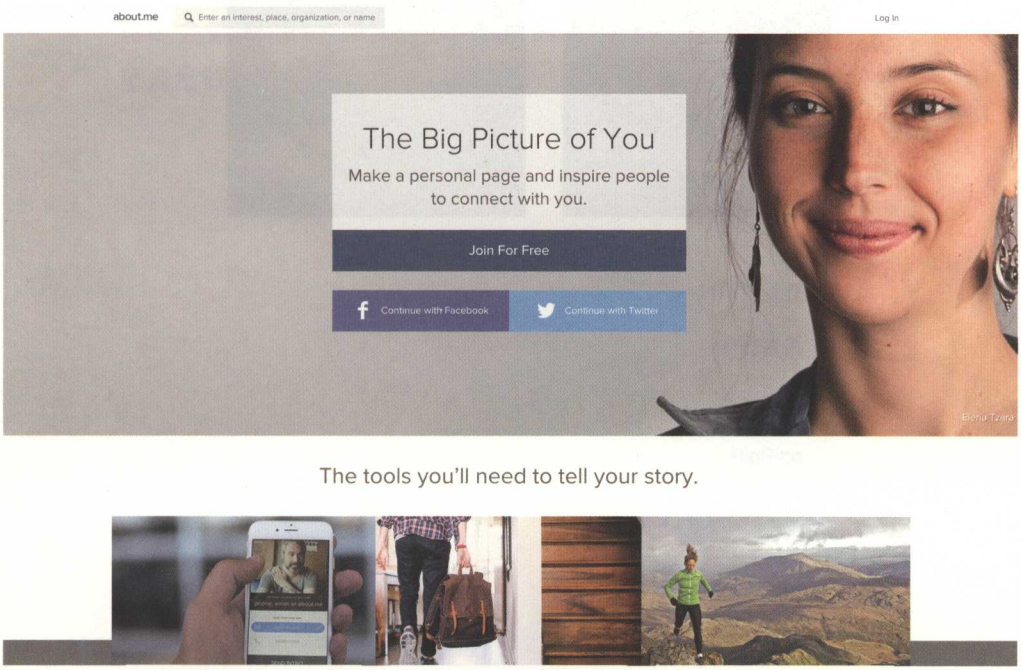
如果作品集有一些动态生成的内容的话，可以选择自己架设服务器并绑定域名，VPS 就是不错的选择。VPS 成本比 GitHub Pages 高，因为需要付费

和配置环境，但是最终跟 GitHub Pages 的效果是类似的。

最后我想说的是，任何作品集都需要有一个重点。如果您想重点突出自己某个技能的深度，可以针对这个技能列出大量作品、项目、专栏或者自己的书。如果想突出技能的广度，光列出您的技能集是不能说服人的，还要在自己的 GitHub 上提交各种使用相关技能的项目。如果自由开发者想招揽一些客户的话，漂亮的过往项目是最重要的。

作品集不一定是严谨而无趣的，曾经有一个前端开发者就将自己的作品集用一个 HTML5 游戏包装起来，让人印象非常深刻。

看到这里，您也许会说，有一些社交网络可以直接生成相关的作品集，比如 LinkedIn、about.me 等。但我的观点是，既然身为一个全栈工程师，那么花一点时间做一些特别的东西会更有趣，不是吗？



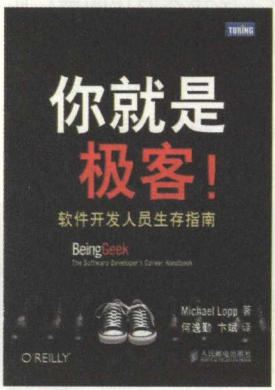
通过 about.me 可以生成自己的作品集，截图来自 about.me。

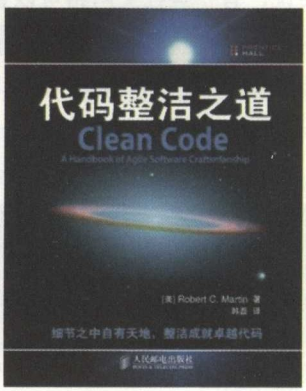
通过社会化媒体，树立起个人的品牌，即使不拿名片出去，也有人知道自己，这才是应该努力的方向。有人说过，“人到三十，不要去找工作，要让工作来找自己”，大概也是这个意思。

延伸阅读

1. 《您就是极客：软件开发人员生存指南》（美）Michael Lopp，人民邮电出版社

2. 《代码整洁之道》（美）Robert C. Martin，人民邮电出版社





全栈工程师眼中的 HTTP

HTTP，是 Web 工程师每天打交道最多的一个基本协议。很多工作流程、性能优化都围绕 HTTP 协议来进行，但是我们对 HTTP 的理解是否全面呢？如果前端工程师和后台工程师坐在一起玩捉鬼游戏，他们对 HTTP 的描述可能会截然不同，从这两个角色的视角看过去，HTTP 呈现出截然不同的形态。

HTTP 简介

超文本传输协议（HyperText Transfer Protocol，HTTP）是互联网上应用最为广泛的一种网络协议。设计 HTTP 的最初目的是提供一种发布和接收 HTML 页面的方法。

OSI 模型[1] 定义了整个世界计算机相互连接的标准，总共分为 7 层，其中最上层（也就是第 7 层）就是应用层，HTTP、HTTPS、FTP、TELNET、SSH、SMTP 和 POP3 都属于应用层。这是软件工程师最关心的一层。

OSI 模型越靠近底层，就越接近硬件。在 HTTP 协议中，并没有规定必须使用它或它支持的层。事实上，HTTP 可以在任何互联网协议或其他网络上实现。HTTP 假定其下层协议提供可靠的传输，因此，任何能够提供这种保证的协议都可以被其使用，也就是其在TCP/IP协议族使用TCP作为其传输层。

OSI 模型				
	数据单元	层	功能	
主机层	Data（数据）	7. 应用层	网络进程到应用程序。	HTTP协议
		6. 表示层	数据表示形式，加密和解密，把机器相关的数据转换成独立于机器的数据。	
		5. 会话层	主机间通讯，管理应用程序之间的会话。	
	Segments（数据段）	4. 传输层	在网络的各个节点之间可靠地分发数据包。	TCP协议
媒介层	Packet/Datagram（数据包/报文）	3. 网络层	在网络的各个节点之间进行地址分配、路由和（不一定可靠地）分发报文。	IP协议
	Bit/Frame（数据帧）	2. 数据链路层	一个可靠的点对点数据直链。	
	Bit（比特）	1. 物理层	一个（不一定可靠的）点对点数据直链。	

OSI 模型，图片来自维基百科。

关于 HTTP 版本

HTTP 已经演化出了很多版本，它们中的大部分都是向下兼容的。客户端在请求的开始告诉服务器它采用的协议版本号，而后者则在响应中采用相同或者更早的协议版本。

当前应用最广泛的 HTTP 版本为 HTTP/1.1，它自从 1999 年发布以来，距

1 开放式系统互联通信参考模型（Open System Interconnection Reference Model），简称为 OSI 模型（OSI model）。

写作本书时已有 16 年的时间。比起 HTTP/1，它增加了几个重要特性，比如缓存处理（在下一章介绍）和持续连接，以及其他一些性能优化。

2015 年 2 月，HTTP/2 正式发布。新的 HTTP 版本有一些重大更新，除了一如既往地向下兼容 HTTP/1 以外，还有一些优化，比如减小网络传输延迟，并简化服务器向浏览器传输内容的过程。主流的服务器（Apache、Nginx 等）和浏览器（Firefox、Chrome、Safari 以及 iOS 和 Android 的浏览器等）的最新版都已经支持 HTTP/2，剩下的就需要网站管理员把服务器升级到最新版了。

例子

下面是一个 HTTP 客户端与服务器之间会话的例子，运行于 www.google.com，端口 80。

客户端首先发出请求。

```
GET / HTTP/1.1
Host:www.google.com
```

第一行指定方法、资源路径、协议版本。当然这是一个简化后的例子，实际请求中还会有当前 Google 登录账户的 cookie、HTTPS 头、浏览器接受何种类型的压缩格式和 UA[1] 代码等。

服务器随之应答。

```
HTTP/1.1 200 OK
Content-Length: 3059
Server: GWS/2.0
Date: Mon, 20 Apr 2015 20:30:45 GMT
Content-Type: text/html
Cache-control: private
Set-cookie: PREF=ID=73d4aef52e57bae9:TM=1042253044:LM=1042253044:S=
SMCc_HRPCQiqy
X9j; expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com
Connection: keep-alive
```

1 用户代理（User-Agent），是指一串字符，表明了当前用户使用什么样的代理在访问站点。浏览器是最常见的一种用户代理。

在这一串 HTTPS 头之后，会紧跟着一个空行，然后是 HTML 格式的文本组成的 Google 主页。

介绍完关于 HTTP 的基本知识，我们来分别看看前端工程师和后台工程师分别是怎样看待这个最熟悉的小伙伴的。

前端视角

前端工程师的职责之一是，让网站又快又好地展现在用户的浏览器中。

从这个角度来说，对HTTP的理解是这样的：打开HttpWatch[1]，然后随意访问一个网站。HttpWatch 会按照浏览器请求的次序，列出打开这个网站的时候发生的请求细节。

- 发出的请求列表。
- 每个请求的开始时间。
- 每个请求从开始到结束花费的时间。
- 每个请求的类型（比如是文本、CSS、JS，还是图片或者字体等）。
- 每个请求的状态码（比如是 200、还是 from cache、304、404 等）。
- 每个请求产生的流量消耗。
- 每个请求 gzip[2] 压缩前的体积，以及在本地 gzip 解压后的体积。

通过查看站点的 HTTP 请求信息，可以得到很多优化信息。每一个前端工

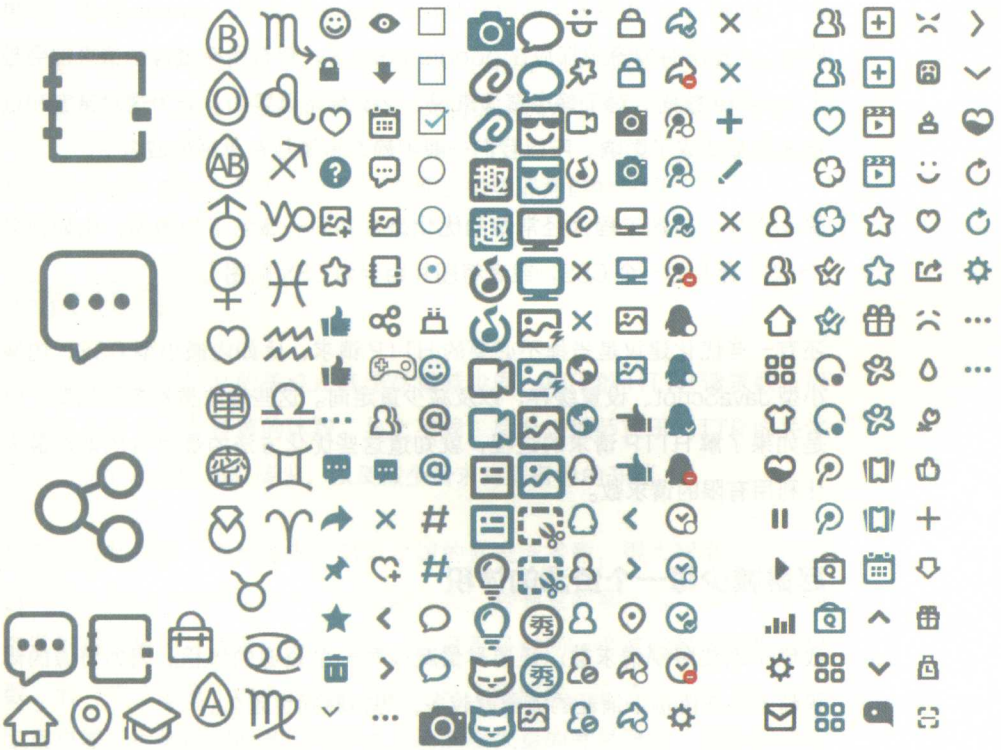
1 HttpWatch 是一个浏览器插件，它可以用来检测页面中所有 HTTP 请求。类似的工具还有 Fiddler，或者各种现代浏览器的开发者工具中的“网络”标签页。

2 gzip 是一种开源的数据压缩算法，其中 g 代表免费的意思（gratis）。HTTP/1.1 协议允许客户端选择要求从服务器下载压缩内容，gzip 是绝大多数客户端和服务器都支持的压缩算法，它在压缩文本文件（比如 HTML、CSS、JavaScript）时压缩效果很好。

程师都知道的基本优化方法是：尽量减少同一域下的 HTTP 请求数，以及尽量减少每一个资源的体积。[1]

尽量减少同一域下的 HTTP 请求数

浏览器常常限定了对同一域名发起的并发连接数的上限。IE6/7 和 Firefox2 的设计规则是，同时只能对一个域名发起两个并发连接。新版本的各种浏览器普遍把这一上限设定为 4 至 8 个。如果浏览器需要对某个域进行更多的连接，则需要在用完了当前连接之后，重复使用或者重新建立 TCP 连接。



QQ 空间的 CSS 贴图由程序自动生成，保证最佳的图片质量、最合理的图片摆放和最小的体积。

1 通过 Chrome 开发者工具中的 PageSpeed 工具，可以快速获得关于站点性能优化的建议。

由于浏览器针对资源的域名限制并发连接数，而不是针对浏览器地址栏中的页面域名，所以很多静态资源可以放在其他域名下（不同的子域名也被认为是不同的域名）。如果您只有一台服务器，可以把这些不同的域名同时指向一个 IP，也就提高了对这台服务器的并发连接数限制（不过要小心服务器压力过大）。

把静态资源放在非主域名下，这种做法除了可以增加浏览器并发，还有一个好处是，减少 HTTP 请求中携带的不必要的 cookie 数据。cookie 是某些网站为了辨别用户身份而储存在用户浏览器中的数据。cookie 的作用域是整个域名，也就是说如果某个 cookie 存放在 google.com 域名下，那么对于 google.com 域名下的所有 HTTP 请求头都会带上 cookie 数据。如果 Google 把所有的资源都放在 google.com 下，那么所有资源的请求都会带上 cookie 数据。对于静态资源来说，这是毫无必要的，因为这对带宽和链接速度都造成了影响。所以我们一般把静态资源放在单独的域名下。

除此之外，前端工程师经常做的优化是合并同一域名下的资源，比如把多个 CSS 合并为一个 CSS，或者将图片组合为 CSS 贴图[1]。

还有一些优化建议是省掉不必要的 HTTP 请求，比如内嵌小型 CSS、内嵌小型 JavaScript、设置缓存，以及减少重定向。这些做法虽然各不相同，但是如果了解 HTTP 请求的过程，就知道这些优化方法的最终目的都是最大化利用有限的请求数。

尽量减少每一个资源的体积

我们不光要限制请求数，还要尽量减少每一个资源的体积。因为资源的体积越大，在传输中消耗的流量就越多，等待时间也越久。

在面试应聘者的时候，我会问的一个基础题目是“常用的图片格式有哪些，它们的使用场景是什么”。如果能选择合适的图片格式，就能够用更小的

1 这种做法被称为 sprite image。

体积，达到更好的显示效果。对图片格式的敏感，能反映出工程师对带宽和速度的不懈追求。

此外，对于比较大的文本资源，必须开启 gzip 压缩。因为 gzip 对于含有重复“单词”的文本文件，压缩率非常高，能有效提高传输过程。

对于一个 CSS 资源的请求耗时，我想说明两个细节。

- 这个 CSS 资源请求的体积是 36.4KB（这是 gzip 压缩过的体积），解压缩之后，CSS 内容实际上是 263KB，可以算出压缩后体积是原来的 13.8%。
- 整个连接的建立花费了 30% 的时间，发出请求到等待收到第一个字节回复花费了 20% 的时间，下载 CSS 资源的内容花费了 50% 的时间。

如果没有设置 gzip，下载这个 CSS 文件会需要好几倍的时间。

后台视角

前端工程师对 HTTP 的关注点在于尽量减少同一域下的 HTTP 请求数，以及尽量减少每一个资源的体积。与之不同，后台工程师对于 HTTP 的关注在于让服务器尽快响应请求，以及减少请求对服务器的开销。

后台工程师知道，浏览器限定对某个域的并发连接数，很大程度上是浏览器对服务器的一种保护行为。浏览器作为一种善意的客户端，为了保护服务器不被大量的并发请求弄得崩溃，才限定了对同一个域的最大并发连接数。而一些“恶意”的客户端，比如一些下载软件，它作为一个 HTTP 协议客户端，不考虑到服务器的压力，而发起大量的并发请求（虽然用户感觉到下载速度很快），但是由于它违反了规则，所以经常被服务器端“防范”和屏蔽。

那么为什么服务器对并发请求数这么敏感?

虽然服务器的多个进程看上去是在同时运行，但是对于单核 CPU 的架构来说，实际上是计算机系统同一段时间内，以进程的形式，将多个程序加载到存储器中，并借由时间共享，以在一个处理器上表现出同时运行的感觉。由于在操作系统中，生成进程、销毁进程、进程间切换都很消耗 CPU 和内存，因此当负载高时，性能会明显降低。

提高服务器的请求处理能力

在早期系统中（如 Linux 2.4 以前），进程是基本运作单位。在支持线程的系统（Linux2.6）中，线程才是基本的运作单位，而进程只是线程的容器。由于线程开销明显小于进程，而且部分资源还可以共享，因此效率较高。

Apache 是市场份额最大的服务器，超过 50% 的网站运行在 Apache 上。Apache 通过模块化的设计来适应各种环境，其中一个模块叫做多处理模块（MPM），专门用来处理多请求的情况。Apache 安装在不同系统上的时候会调用不同的默认 MPM，我们不用关心具体的细节，只需要了解 Unix 上默认的 MPM 是 prefork。为了优化，我们可以改成 worker 模式。

prefork 和 worker 模式的最大区别就是，prefork 的一个进程维持一个连接，而 worker 的一个线程维持一个连接。所以 prefork 更稳定但内存消耗也更大，worker 没有那么稳定，因为很多连接的线程共享一个进程，当一个线程崩溃的时候，整个进程和所有线程一起死掉。但是 worker 的内存使用要比 prefork 低得多，所以很适合用在高 HTTP 请求的服务器上。

近年来 Nginx 越来越受到市场的青睐。在高连接并发的情况下，Nginx 是 Apache 服务器不错的替代品或者补充：一方面是 Nginx 更加轻量级，占用更少的资源和内存；另一方面是 Nginx 处理请求是异步非阻塞的，而 Apache 则是阻塞型的，在高并发下 Nginx 能保持低资源、低消耗和高性能。

由于 Apache 和 Nginx 各有所长，所以经常的搭配是 Nginx 处理前端并发，Apache 处理后台请求。

值得一提的是，新秀 Node.js 也是采用基于事件的异步非阻塞方式处理请求，所以在处理高并发请求上有天然的优势。

DDoS 攻击

DDoS 是 Distributed Denial of Service 的缩写，DDoS 攻击翻译成中文就是“分布式拒绝服务”攻击。

简单来说，就是黑客入侵并控制了大量用户的计算机（俗称“肉鸡”），然后在这些计算机上安装了 DDoS 攻击软件。我们知道浏览器作为一种“善意”的客户端，限制了 HTTP 并发连接数。但是 DDoS 就没有这样的道德准则，每一个 DDoS 攻击客户端都可以自由设置 TCP/IP 并发连接数，并且连接上服务器之后，它不会马上断开连接，而是保持这个连接一段时间，直到同时连接的数量大于最大连接数，才断开之前的连接。

就这样，攻击者通过海量的请求，让目标服务器瘫痪，无法响应正常的用户请求，以此达到攻击的效果。

对于这样的攻击，几乎没有什么特别好的防护方法。除了增加带宽和提高服务器能同时接纳的客户数，另一种方法就是让首页静态化。DDoS 攻击者喜欢攻击的页面一般是会对数据库进行写操作的页面，这样的页面无法静态化，服务器更容易宕机。DDoS 攻击者一般不会攻击静态化的页面或者图片，因为静态资源对服务器压力小，而且能够部署在 CDN 上。

这里介绍的只是最简单的 TCP/IP 攻击，而 DDoS 是一个概称，具体来说，有各种攻击方式，比如 CC 攻击、SYN 攻击、NTP 攻击、TCP 攻击和 DNS 攻击等。

BigPipe

前端跟后端在 HTTP 上也能有交集，BigPipe 就是一个例子。

现有的 HTTP 数据请求流程是：客户端建立连接，服务器同意连接，客户端发起请求，服务器返回数据，客户端接受并处理数据。这个处理流程有两个问题。

现有的阻塞模型，黄色代表服务器生成页面，白色代表网络传输，紫色代表浏览器渲染页面。

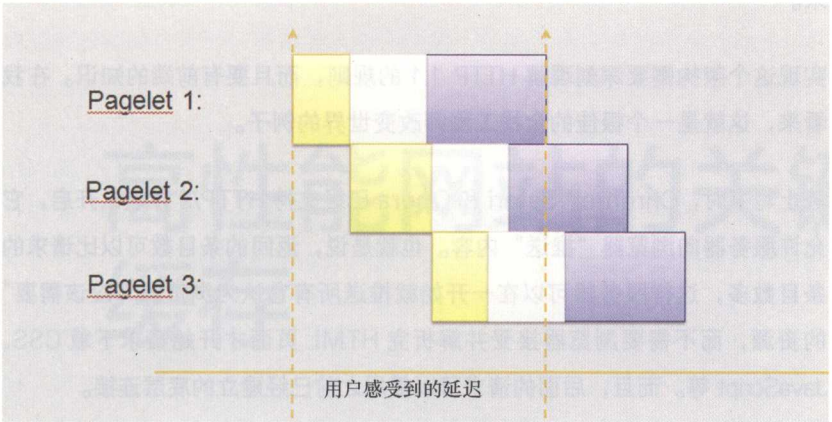
第一，HTTP 协议的底层是 TCP/IP，而 TCP/IP 规定 3 次握手才建立一次连接。每一个新增的请求都要重新建立 TCP/IP 连接，从而消耗服务器的资源，并且浪费连接时间。对于几种不同的服务器程序（Apache、Nginx 和 Node.js 等），所消耗的内存和 CPU 资源也不太一样，但是新的连接无法避免，没有从本质上解决问题。

第二个问题是，在现有的阻塞模型中，服务器计算生成页面需要时间。等服务器完全生成好整个页面，才开始网络传输，网络传输也需要时间。整个页面都完全传输到浏览器中之后，在浏览器中最后渲染还是需要时间。三者是阻塞式的，每一个环节都在等上一个环节 100% 完成才开始。页面作为一个整体，需要完整地经历 3 个阶段才能出现在浏览器中，效率很低。

BigPipe 是 Facebook 公司科学家 Changhao Jiang 发明的一种非阻塞式模型，这种模型能完美解决上面的两个问题。

通俗来解释，BigPipe 首先把 HTML 页面分为很多部分，然后在服务器和浏览器之间建立一条管道（BigPipe 就是“大管道”的意思），HTML 的不同部分可以源源不断地从服务器传输到浏览器。BigPipe 首先输送的内容是框架性 HTML 结构，这个框架结构可能会定义每个 Pagelet 模块的位置和

宽高，但是这些 pagelet 都是空的，就像只有钢筋混泥土骨架的毛坯房。



BigPipe 页面的渲染流程。

服务器传输完框架性 HTML 结构之后，对浏览器说："我这个请求还没结束，我们保持这个连接不要断开，不过您可以先用我给您的这部分来渲染。"

所以浏览器就开始渲染这个"不完整的 HTML"，毛坯房页面很快出现在用户眼前，具体的页面模块都显示"正在加载"。

接下来管道里源源不断地传输过来很多模块，这时候最开始加载在服务器中的 JS 代码开始工作，它会负责把每一个模块依次渲染到页面上。

在用户的感知上，页面非常快地出现在眼前，但是所有的模块都显示正在加载中，然后主要的区域（比如重要的用户动态）优先出现，接下来是 logo、边栏和各种挂件等。

为什么 BigPipe 能够让服务器对浏览器说"我这个请求还没结束，我们保持这个连接不要断开"呢？答案是 HTTP1.1 的分块传输编码。

HTTP 1.1 引入分块传输编码，允许服务器为动态生成的内容维持 HTTP 持久链接。如果一个 HTTP 消息（请求消息或应答消息）的 Transfer-Encoding 消息头的值为 chunked，那么消息体由数量不确定的块组成——

也就是说想发送多少块就发送多少块——并以最后一个大小为 0 的块为结束。

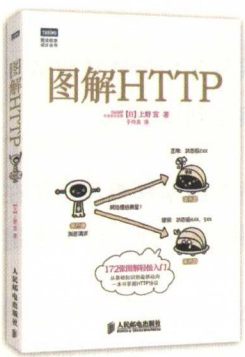
实现这个架构需要深刻理解 HTTP 1.1 的规则，而且要有前端的知识。在我看来，这就是一个极佳的全栈工程师改变世界的例子。

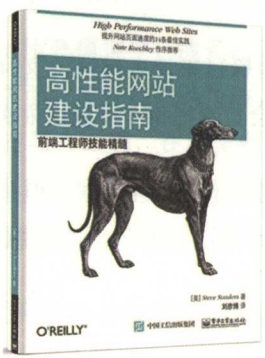
截止写书时，Chrome、Safari 和 Opera 已经支持 HTTP/2 并默认开启，它允许服务器向浏览器“推送”内容。也就是说，返回的条目数可以比请求的条目数多，这样服务器可以在一开始就推送所有它认为浏览器“应该需要”的资源，而不需要浏览器接受并解析完 HTML 页面才开始请求下载 CSS、JavaScript 等。而且，后面的请求可以复用之前已经建立的底层连接。

延伸阅读

1. 《图解 HTTP》（日）上野宣，人民邮电出版社

2. 《高性能网站建设进阶指南》（美）Steve Souders，电子工业出版社





高性能网站的关键：缓存

计算机科学中最无奈的两件事是缓存失效和命名。

——Phil Karlton

什么是缓存

Phil Karlton 的这句话很有意思，我们一般人会认为，计算机科学的难题数不胜数，比如在算法或者复杂性理论方面都有很多尚待解决的问题。那么为什么这位前网景通讯公司（Netscape）的知名工程师要这么说呢?

这是可能是因为，复杂性理论方面的难题，可能最终还是有解的。而缓存失效是分布式系统中最常见，也几乎没有最优解决方案的难题。

我本来想在这一章蜻蜓点水地介绍一下“高性能网站”各个方面的话题，但是我发现这不可能。“高性能网站”足以写成厚厚的一本书，而全栈工程师面对的是一片大海，如果我能描绘通往大海的一条小河，并且引起您的兴趣，我的目的就达到了。

缓存对于站点性能起到举足轻重的作用，很多时候，优化算法和压缩图片带来的优化效果可能远远不如优化缓存。

我在西安电子科技大学读书的时候，当时图书馆楼还在新建，所以图书馆临时设立在自习楼的五、六两层。图书馆的入口在五层，大部分书按照分类放在六楼的不同房间。但是有趣的一点是，被归还的书会暂时放在五楼入口旁边的一个房间，因为这些书曾被借阅，因此就很有可能再次被借阅者挑选出来。所以这些书就不用浪费时间放回六楼的原位了，我也会常常去这个房间看看有哪些热门图书。

它给我的启发是，这个房间就像是图书馆的缓存。因为计算机系统中的缓存有这样几种功效。

- 存储频繁访问的数据（这里的数据是图书）。
- 内存缓存减少磁盘 I/O（不用到 6 楼去找书）。
- 保存耗时的操作，以便下次使用（找书和整理书是耗时的操作）。

下面我来谈谈在一个 Web 站点中，它的数据流从服务器端到浏览器端，哪些地方可以使用缓存来优化。

服务器缓存

对于一些计算量大的 Web 服务、服务器内存或 CPU 等性能不好，或者像一些独立开发者跟其他人共享虚拟服务器（因此只能得到部分内存和 CPU）的时候，服务器的计算时间可能占整个页面响应时间的很大一部分。这种情况下，优化服务器端的缓存就尤为重要了。

基本的数据库查询缓存

我们从服务器到客户端，依次来讲解缓存的作用，首先从数据库开始。

对于大型网站来说，数据库里的数据量往往是非常大的，而对于数据的查询又是比较耗时的操作，所以我们可以开启 MySQL 查询缓存来提高速度，并且减少系统压力。MySQL 默认不开启查询缓存，但我们可以通过修改 MySQL 安装目录中的 my.ini 来设置查询缓存。设置的时候可以根据实际情况配置缓冲区大小、单个查询的缓冲区大小等。

如果您希望优化 MySQL 服务器的查询性能和速度，可以在 MySQL 配置中增加这两项。

```
query_cache_size=SIZE
```

SIZE 是指为查询缓存开辟多大的空间。默认是 0，也就是禁用查询缓存。

```
query_cache_type=OPTION
```

设置查询缓存的类型，可选的值有以下这三种。

- 0：不将查询结果缓存起来，也不从缓存中获取查询结果。

- 1：所有的缓存结果都缓存起来，除非查询命令以 SELECT S_NO_CACHE 开始。
- 2：只缓存查询命令以 SELECT SQL_CACHE 开始的查询结果。

具体的设置方法不是我们讨论的重点，重点是要了解适合设置查询缓存的场景。因为每一次 select 查询的结果都会被缓存起来，如果数据库数据没有发生变化[1]，下一次查询就会直接从缓存里返回数据。但是如果数据库发生了变化，那么所有与该表有关的查询缓存全部失效。

所以，对于查询操作远远多于修改操作的数据库，开启数据库查询缓存是很有益的；但是对于修改操作很多的数据库，由于缓存经常会失效，就起不到加速的效果。不仅如此，由于数据库要花费时间写缓存，所以实际上速度更慢了。

这个问题就是“缓存命中率不高”，所以配置缓存之后第一件事就是查询命中率，如果命中率低，不如不做缓存。

这里需要注意的是，两次 SQL 文本必须完全相同。如果前后两次查询使用了不同的查询条件，就会重新查询。如第一次查询时没有输入 where 条件语句，后来发现数据量过多，于是利用 where 条件过滤查询的结果，此时即使最后的查询结果是相同的，系统仍然是从数据文件中获取数据，而不是从缓存结果中。再如，select 后面所使用的字段名称也必须是相同的。如果查询语句中有一个字段名称不同，或者前后两次查询所使用的字段数量不同，都会被系统认为是不同的 SQL 语句，需要重新解析并查询。

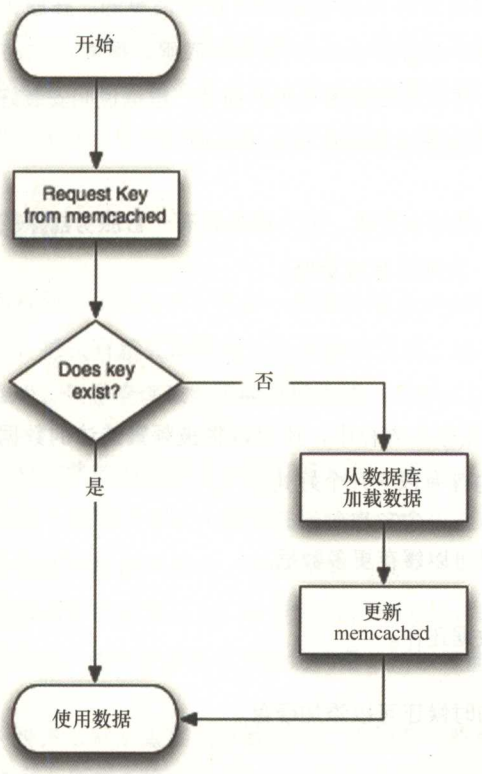
扩展数据库缓存：memcached

MySQL 的自带缓存有一个问题，它的缓存池大小是在 MySQL 所在服务器上开辟，能使用的内存空间是有限的。在比较大型的网站中，缓存就不够用了，这时候需要使用服务器集群来实现数据库缓存。

1 没有运行 INSERT/UPDATE/DELETE/MERGE 等操作的话，数据库就不会变化。

memcached 应运而生，它是一个高性能分布式内存对象缓存系统，用于减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。memcached 可以与数据库查询缓存配合使用，查询流程如下图所示。

您也许发现了数据库查询缓存的一个设计原则：**其缓存失效设计是很粗糙的**。只要某个表发生了更新操作，所有对这个表的查询都会失效。这是为了保证数据的时效性而降低了数据的命中率。



memcached 一般查询流程。

memcached 的缓存失效与此不同，**它采用的是按时间来过期的设计**。memcached 相当于应用程序和数据库之间的中间层，通过网络 API 设置和

调用。memcached 储存的是名值对，而且设置了一个过期时间，只要过期时间没有到，应用程序就会从 memcached 中获取数据。这时候即使发生了数据库更新操作，缓存的查询结果也仍然是之前保存的旧数据，直到设置的时间过期。这样提高了缓存的性能，带来的影响就是，数据可能是“不新鲜”的。

memcached 支持集群，而且有诸多优点，所以可以有效利用多台机器的内存，提高命中率。

如果您使用 WordPress，那么开启 memcached 是很简单的。在服务器安装好 memcached 后，再去 WordPress 的插件列表里，搜索 cache 或 memcached 之类的关键词，可以找到很多相关的插件。根据说明安装好这些插件后，一般就可以无缝衔接缓存软件和 WordPress 了。

但是 memcached 也不是总是那么有效，因为如果只有一台服务器，就用不到它的服务器集群的优势，反而让系统更慢。

再加一层文件缓存

除了可以将数据库查询结果缓存在内存中，还可以将被频繁造访的数据缓存在文件中。文件 I/O 比起内存有以下几个好处。

- 硬盘容量比内存大，所以可以缓存更多数据。
- 数据更安全，断电之后数据还在。
- 易于扩展，硬盘不够用的时候还可以添加硬盘。

但是文件缓存没有内存缓存快，只能作为内存缓存的补充，在获取数据时，先从最快的地方读取，如果没有就继续往后找。查找优先级为：内存缓存→文件缓存→数据库。

PHP 框架 CodeIgniter 的数据库缓存类，允许您把数据库查询结果保存在文

本文件中，以减少数据库访问。

如果激活了 CodeIgniter 的缓存特性，那么在某页面首次被加载时，数据库查询的结果对象将会被序列化，并保存在服务器的文本文件中。而此页面再次被加载时，缓存文件将会替代数据库查询。如此，在被缓存的页面中，您的数据库使用率会降至零。

只有读类型查询会被缓存，因为只有这种查询会产生结果集。 而写类型查询，因为不会产生结果集，故缓存系统不对之进行缓存。

缓存文件不会过期，除非您删掉它，否则任何被缓存了的查询会一直存在。缓存系统允许您按页面清除，或把所有缓存都清除掉。一般来说，您可以在某些事件（比如向数据库添加了数据）发生时用特定的函数来清除缓存。

静态化

有两种静态化的方法，其中一种是类似 WordPress 的静态化插件，安装很简单，每次有新文章就自动生成静态页面。这种方法还是将数据保存在数据库中，只是会读取数据库之后生成一些静态页。

这一种方法的原理跟文件缓存很相似。静态化页面之后，服务器每次接收到对这个页面的请求，都会直接给出这个页面的静态文件，所以就省略了后台运算和数据库查询。优点是能大大加快访问速度，同时减轻服务器处理大量请求的运算压力。在前面我们也说过，因为静态化的页面对服务器的压力小，能有效承担巨大的访问量，所以还能抵御 DDoS 攻击。

另一种方法就是直接抛弃数据库。比如有一些博客作者会用 Jekyll 系统来写博客，将整个博客站点静态化。完全抛弃数据库的好处是，可以将生成的静态网页直接托管在静态资源站点，比如 GitHub Pages 或者 Amazon S3，而不用操心数据库服务器的问题，不光整个系统稳定很多，费用上也更加低廉（GitHub 更是完全免费的，而且提交 Markdown 源代码后可以让它在服务器端生成站点）。

对于完全静态化的站点，可以采用第三方插件来支持用户生成内容。比如 Disqus 就是一个第三方的评论插件，通过 JavaScript 代码插入到静态页中。用户的评论数据都储存在 Disqus 的服务器上，对我们是透明的，很方便。

值得一提的是，我们从 URL 是无法判断后台是否真正静态化的。对于一个 URL 为 /blog/index.html 的页面，也有可能是 PHP 页面通过 UrlRewrite 来改写的。通过 Apache 或者 Nginx 可以将一个对静态资源的请求（index.html）转给一个动态应用程序（比如 PHP）来处理。

浏览器缓存

前面说的缓存都是发生在服务器端的，适用的情况是那些服务器性能为主要瓶颈的场合，通过缓存来将一个长的计算时间跳过，起到提高性能的作用。而当浏览器访问一个站点的时候，网络连接是主要瓶颈，我们可以通过设置浏览器缓存来跳过 HTTP 请求。

如果在浏览器设置缓存，通常有两个主要作用。

- 对用户来说，减少请求可以更快地加载页面，节省流量。如果用户是在手机上用 3G 或 4G 访问页面，这一点就很关键。
- 对网站来说，减少带宽压力和费用。假设有 1 亿的访问量，如果能把大小为 10KB 的 CSS 缓存起来，可以节省不小的开支。

对于浏览器来说，如何缓存一个资源是服务器端制定的策略，自己只是根据服务器的“指令”来执行而已。服务器的“指令”就是前面介绍过的 HTTPS 头。

服务器通过对每个资源的 HTTP 响应头设置属性和值，来发出自己的缓存指令。主要会有两种缓存指令，我们以一个图片 image.png 为例。

第一种：Expires

对于一个普通的请求，服务器可能会说："您拿着这个资源吧，直到 2020 年您都别再向我要了。"

```
Expires: Thu, 15 Apr 2020 20:00:00 GMT
```

那么浏览器如果再次命中对这个资源的需求，就不会再去发起 HTTP 请求，而是直接从缓存（在硬盘中）读取。

```
200(from cache)
```

这种缓存是最快的，因为没有任何 HTTP 请求发生。当用户需要这个资源，浏览器就直接从缓存中读取，不再需要询问服务器端的意见（服务器端甚至不知道您在浏览 image.png）。所以 HttpWatch 是推荐对所有的静态资源都设置 Expires。

第二种：Last-Modified

对于一些有可能过期的请求，服务器可能会比较慎重地说："您拿着这个资源吧，这个资源上次修改时间是 2014 年 3 月 1 日，如果用户要用，您就问问我改变了没，或者直到 2014 年 12 月 31 日文件自动过期。"

```
Last-Modified: Tue, 01 Mar 2015 03:42:36 GMT
```

那么浏览器如果在 2015 年 3 月 10 日访问了 image.png，就会将这个图缓存在硬盘中。过了几天，浏览器又命中了对这个资源的需求，就会发起一个 HTTP 请求。

在 HTTPS 头中，浏览器问："我这里有个 image.png，它的最后修改时间是 2015 年 3 月 1 日，现在用户又要了，您那个文件有过更新没？"

```
If-Modified-Since:  Tue, 01 Mar 2015 03:42:36 GMT
```

服务器如果回答："没更新，您直接用吧。"这个回答中就不需要带上请求的文件体了，只用一个 HTTPS 头表示文件未更新即可。304 就是这句话的

代号，代表资源未修改的意思。

```
304
```

另一种情况是，image.png 后来更新过了，服务器就会说："更新过了，我现在给您一个新的图片。"然后就正常返回请求文件（200），并且把整个图片作为 HTTP 正文发送给浏览器。

通过这种缓存方式，无论资源是否发生了更新，仍然至少会发生一来一去 HTTPS 头的传输和接收，所以速度比不上 Expires。

从服务器端的角度来看，有时候我们并不希望对静态资源的请求中大部分都返回 304。因为这可能说明我们的很多用户都在频繁访问站点，而且我们的资源很少更新，就好像它们一直问"资源修改了吗？"，我们一直回答"没有修改"。这里可以使用 Expires 来设置过期时间，这样它们就不会"烦我们"了。对于服务器管理员来说，保持 304 为一个合理的比例即可。我们可以通过查看服务器的 log，查看 304 响应与 200 响应的比例，来做出一个合理的缓存策略。

Restful Web API

表征性状态传输（Representational State Transfer，REST）是 Roy Fielding 博士在 2000 年发表的博士论文中提出来的一种软件架构风格。

目前，在 3 种主流的 Web 服务实现方案中，因为 REST 模式最简洁，也能合理地利用 HTTP 操作的语义，所以越来越多的 Web 服务开始采用 REST 风格设计和实现。比如，Amazon.com 提供接近 REST 风格的 Web 服务进行图书查找。

Restful 的目的是定义如何正确地使用 Web 标准，优雅地使用 HTTP 本身的特性。原则上是对资源、集合、服务（URL）、get、post、put、delete（操作）的合理使用。

举例来说，如果请求一个资源，但是服务器上没有这个资源，这时候就应该对 HTTPS 头设置 404，而不是设置 200。

HTTP 1.1 加入的 Cache-Control

其实 Expires 跟 Last-Modified 已经能满足我们绝大多数需求了，但是 HTTP1.1 又新增了一个属性 Cache-Control，它的功能跟 Expires 类似，不过有更多的选项。

Expires 的值是一个日期，表示某日期之前都不再询问。

Cache-Control 的值是 :max-age=7776000，max-age 的单位是秒，从浏览器接收到文件之后开始计时。

如果您不知道怎么判断，就只用 Expires，或者（为了兼容某些老客户端）同时设置 Expires 和 Last-Modified。

如果 topMenu.js 设置了 Expires 直到 2020 年都不过期，那么怎么让客户端知道我们修改了 topMenu.js 呢?

答案是修改 Query String。按照规范，Query String 是 URL 中的一个部分，比如 `http://server/program/path/?query_string` 问号后面的字符串就是 Query String。

按照 HTTP 规范，如果修改了请求资源的 Query String，就应该被视为一个新的文件。

这个 Query String 可以被服务器端 CGI 或者应用程序理解，而且可以设置多个名值对（比如 ?foo=1&bar=2）。与缓存相关的一点是，如果 Query String 发生了改变，则被视为 URL 发生了改变。这时候，浏览器会认为这是一个新的资源。而对于服务器而言，如果有 CGI 或者应用程序捕捉或处理 Query String，就会去处理，如果没有，就简单地忽略 Query String，直接返回资源。

下面是推荐的浏览器缓存设置最佳实践。

- 对于动态生成的 HTML 页面使用 HTTPS 头：Cache-Control: no-cache。
- 对于静态 HTML 页面使用 HTTPS 头：Last-Modified。
- 其他所有的文件类型都设置 Expires 头，并且在文件内容有所修改的时候修改 Query String。

浏览器缓存的现实世界

服务器端可以设置缓存规则，告诉浏览器应该如何遵循和实现，但在服务器不能掌控的地方也许会出现一些意外。

- 缓存会被挤出。
- 文件有可能在运营商服务器上被劫持。

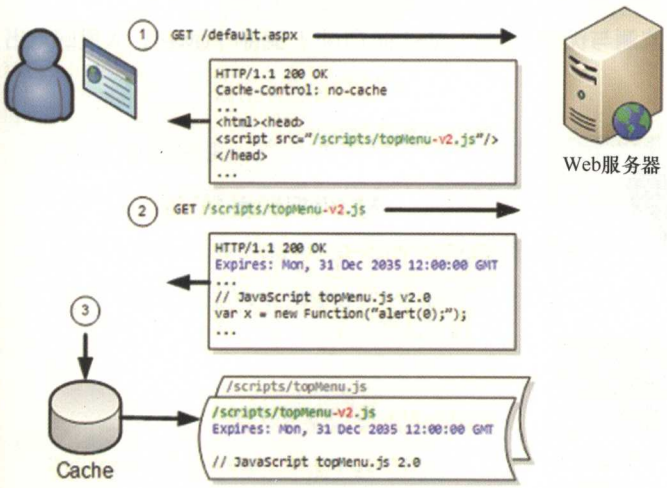
所谓缓存被挤出，是因为浏览器的缓存空间是有限的，所有网站希望缓存的文件都塞在用户硬盘，形成一个先进先出的队列。所以即使设置了 20 年的缓存时间，也基本上不能保证有那么久的生命期，而会在某一个时间点被其他网站设置的缓存挤出硬盘。而且用户也有可能主动清除浏览器缓存，某些系统清理软件也可能清理浏览器缓存。这一点无可厚非，我们从技术层面上也无法解决，大不了让用户重新加载一下资源就好了。

第二个问题是，用户的宽带运营商为了提高速度，可能会在自己某节点服务器上缓存您的文件（比如 style.css?v1），好处是当用户请求这个文件的时候，运营商无需来您的服务器上请求文件，而自己直接就给出了。

问题来了，如果您的 Query String 更新了（style.css?v2），按照 HTTP 规范，这理应被视为一个新的文件，但是运营商仍然可能会拿自己节点的缓存，而不是遵循规范。有点可恶对不对？这就是我们在用户量极大的情况

下侦测到的情况，虽不太常见，但是有可能发生。所以，为了保证更新的文件下发到所有的用户，我们会使用更加强硬的方法：修改文件名，而不是仅仅修改 Query String。

这种流程比较复杂，需要同时修改文件名和引用它的文档里的文件名。在 QQ 空间，我们使用自动化的方式来生成新文件名并修改 HTML 中的引用。



修改资源的文件名，比修改后缀更可靠。

结论

结合上面的分析，这是 QQ 空间静态资源在浏览器端使用的缓存策略。

- 对于动态生成的 HTML 页面使用 HTTPS 头：Cache-Control: no-cache。
- 对于静态 HTML 页面使用 HTTPS 头：Last-Modified。
- 其他所有的文件类型都设置 Cache-Control 头，**并且在文件内容有所修改的时候修改文件名**。

以上就是在 Web 栈流程中比较常见的缓存方面的问题。缓存能否获得性能增益，取决于很多因素。一些因素是关于您的服务器压力、数据库使用情

况和提供的服务类型；另一些因素是关于您的用户如何访问您的网站，以及他们的网络环境。我们作为工程师，只能不断优化和调整策略，往更优的方向去优化。

延伸阅读

1. 《网站性能监测与优化》（美）Alistair Croll / Sean Power，人民邮电出版社

大前端

总体来讲，在计算机程序和系统中，“前端”（front-end）作用于采集和显示信息，“后端”（back-end）进行处理。Web 应用程序和桌面应用程序的界面样式、视觉呈现、用户交互属于前端。

前端工程师

知识体系

岗位细分

前端工程师

我们最常见的 Web 栈中靠近用户输入的那一部分，也就是靠近浏览器的部分，属于前端的范畴。具体来说，浏览器中发生的一切和服务器中涉及输入输出的这一部分，都属于前端工程师的工作职责。

前端工程师主要使用的语言是 HTML、CSS 和 JavaScript，有时候会写一些服务器的页面模板语言（比如 PHP）。

我从 2010 年进入腾讯工作，直到今天，能够明显感受到的一点是，前端发展到今天，已经发生了很大的变化。

在 2010 年，前端开发岗位必须掌握的一项能力是对 IE6 和 IE7 的兼容性。工程师需要手动把图片拼接成 CSS 贴图，CSS 也不经压缩就发布出去。渐渐地，IE 用户越来越少，所以，我们现在已经不把 IE7 以下的浏览器兼容性作为招聘要求。不过由于移动设备的爆炸性增长，现在前端开发岗位开始要求有移动端页面开发的经验，或者熟悉响应式页面开发。Grunt 等发布流程的成熟，也让前端工程师免除了很多枯燥的工作。

总之，变化一直都在发生，这是一个需要终身学习的行业。不夸张地说，如果我一个月没有关注行业动态，就会发现自己已经错过了很多新技术。

知识体系

前端工程师需要涉及的知识面比较广，我大概对我的个人偏好做一下梳理。在招聘初级工程师的时候，我一般会考察应聘者对以下知识的掌握程度。

- 对浏览器兼容性的了解。
- 对 HTML/CSS/JavaScript 语法和原理的理解。

- 对编辑器和插件的熟悉程度。
- 对调试工具的了解程度。
- 对版本管理软件的熟悉和应用经验。
- 对前端库 / 框架的使用。
- 标准 / 规范。

招聘中级工程师时，我一般考察应聘者对以下知识的掌握程度。

- 对代码质量、代码规范的理解。
- 对 JavaScript 单元测试的熟悉。
- 对性能优化的应用和理解。
- 对 SEO 的应用和理解。
- 代码部署。
- 移动 Web。

对高级工程师，除了上面的考察点以外，还会问这些方面的经验。

- 代码架构。
- 安全。
- 对自动化测试的理解。

越接近高级工程师，越考察对某个点的本质理解，以及在项目和团队中的引导作用，而不是对某工具的使用经验。对于上面的这些技术方向，我不会在本章中一一介绍，一个原因是篇幅所限，另一个原因是有一些方向并不只是前端工程师会遇到，而是跟后台工程师的知识体系相通。比如代码

质量、规范、单元测试、性能、版本管理……对于这些话题，我会在单独的章节中详细说明。

易于上手，难于精通

不同于某些“难于上手、难于精通”的职业，前端这一岗位就像暴雪公司的游戏设计一样：“易于上手、难于精通”。

举例而言，HTML 的全称是超文本标记语言，超文本的意思是说，比起我们在记事本中写的普通文本多了一些语义化的信息，比如链接、加粗和标题等。标记语言更加简单，就是用一些标签把普通文本标记起来。标记语言没有逻辑，只是描述性的标签，所以算不上是程序语言。程序语言有的循环判断等逻辑，HTML 都没有。这就是它易于上手的地方。

这是一段最简单的 HTML 代码，但它也是一个完整的页面。

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>
<p>Hello World!</p>
</body>
</html>
```

但是 HTML 又很难。前端工程师对照设计稿还原出页面之后，还要考虑机器是如何理解这个页面的。对于用户而言，视觉上加大加粗就是一个标题；但是对于机器，比如搜索引擎或者盲人使用的读屏软件，是否能理解它是一句标题？这需要我们使用语义化的标签。

有的时候，为了完美地实现设计稿还原，一个视觉上看上去像一个下拉选择器的输入框，我们会使用 a 或者 span 标签加上一些隐藏的列表模块来实现。当用户点击标签的时候，就用 JavaScript 让隐藏的列表模块滑出来。

使用这种方法，我们可以快速地创建出在各浏览器中表现一致的按钮，而且可以轻松地自定义样式，免受各种 bug 困扰，但同时这种“黑”科技也带来了可访问性问题。具体来讲，如何让盲人知道这是一个下拉选择器？这时候可以使用 role 属性来增强这个文档对象模型（DDM）的语义。这需要我们了解 WAI-ARIA 的知识。

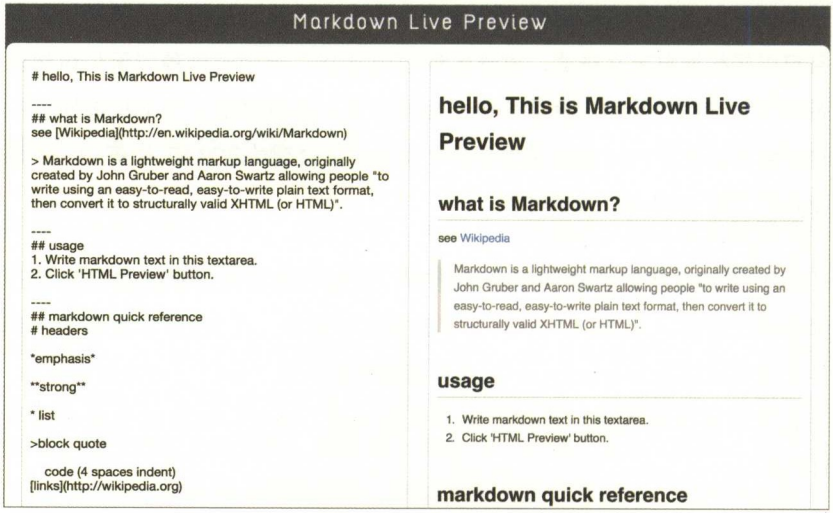
HTML 虽然是比较严格和简单的语言，但有时在写代码和阅读代码的时候效率比较低。John Gruber 为了改变这种现状，在 2004 年发明了一种纯文本格式语法 Markdown[1]。这种语法的目标是“提供一种读起来简单，写起来也简单的语法，并且如果您乐意的话，也可以随时转化成合法的 HTML”。

您手中的这本书，就是使用 Markdown 语言来编写的。它比 Word 更好用的地方是，写作者无需关注字号和样式，只需要设置“一级标题”“二级标题”“三级标题”“加粗”“引用”等语义即可。具体的样式可以在美编阶段统一调整。更美好的是，它不会产生任何无意义的标签，而 Word 经常产生无意义的标签。

Markdown 比 HTML 更易读易写，但是浏览器是不会渲染的，那么就必须进行 Markdown 到 HTML 的转化。开发者可以选择两种转化方式。一种是在开发环境把 Markdown 转化成 HTML，再发布到服务器上，或者直接由服务器自动转化成 HTML 文件（Jekyll 支持这两种转化方式），总之浏览器得到的已经是一个正常的 HTML 页面了。

第二种方式是把包含 Markdown 代码的 HTML 页面发布到服务器上，然后当浏览器下载 HTML 页面之后，页面中的 JavaScript 代码开始把 Markdown 解析成对应的 HTML 代码。一般推荐第一种方式，因为不依赖浏览器端的 JavaScript 运行，可用性更好，也对 SEO 更有帮助。

1 Markdown 这个名字也看成是于对 Markup（标记）的戏谑演变。



在线工具[1]可以实时把 Markdown 转化成 HTML，截图来自 markdownlivepreview.com。

也有人想出另一种方法，发明了 zen-coding 来加快前端工程师的编码速度。zen-coding 现在改名为 emmet，在各大编辑器中都有插件。

前端技术的“易于上手”导致它在某些技术人员那里不受待见。他们认为 HTML 与 CSS 根本都不是程序语言，甚至认为 JavaScript 是一种功能不全的玩具型语言。所以直到我几年前毕业的时候，大学都没有前端相关的课程和专业。而市场对前端工程师的需求又很大，学校的输出跟市场的要求没有对接上，所以往往出现学生找不到工作，公司又招不到人的现状。

我并不是建议直接开设“前端开发”专业，因为前端开发也是软件开发的一个分支，与服务器开发和其他软件开发共享一些基础课程，是所有工程师都需要学习的，比如项目管理、数据库、软件开发流程和 C++ 等。我的建议是，在大三或者大四的方向课程设计上，或者选修课设计上增加与时俱进的前端开发课程，使用业界最新的编程语言去教学，这样对毕业生、对用人单位都是好事。

1 http://markdownlivepreview.com/

框架 vs 库

框架（framework）和库（library）的区别是什么？其实这两个词在不同的语境下，有时候是可以相互替代的。但是严格来说，框架应该是比库更广泛的概念。

一个库是一系列对象、方法等代码，您的应用程序可以把这个库“链接”进来。这个库起到了重用代码的作用，为您省下了重写这部分代码的工作量。

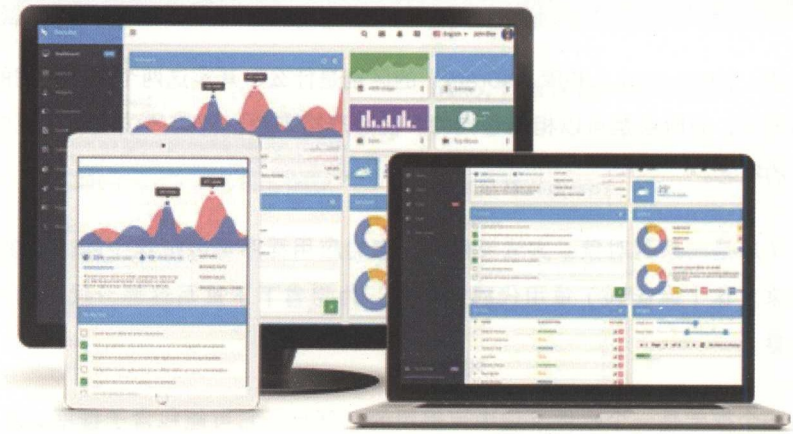
而一个框架是一个软件系统中可重用的一部分。它可能包含子程序、库、胶水语言、图片等一些“资源”，这些资源一起组成了软件项目。框架不像库，可能包含多种语言，某些功能可能通过 API 的方式让主程序调用。

所以框架是一个更加灵活和宽松的名词，在具体的情景中，它可能指一个库、多个库、脚本代码，或者多个可单独运行的子程序的集合。

以前端来举例，jQuery 就是一个库。jQuery 是纯粹的 JavaScript 代码，它的目标是使用更少的代码处理 DOM 相关操作。当我们使用 jQuery 时，相当于引入了新的对象和方法，可以利用 jQuery 定义的代码，不需要重写这部分功能。

而 Sencha 公司的 ExtJS 是一个框架。首先 ExtJS 不仅包含 JavaScript 代码，还包含了 CSS 和图片。其次它的功能是方便开发者搭建可交互的 Web 应用程序，里面有一些完整功能的模块，比如绘制可交互的图表。所以 ExtJS 是一个框架。

作为一个前端工程师，面对的框架和库层出不穷，很容易陷入迷茫，到底应该使用哪种？一个常见的误区是，存在某个强大的“终极方案”，可以解决一切 Web 应用程序开发的问题。经常有一些人请我推荐一些好用的前端框架，我不知道如何回答。



ExtJS 框架，截图来自 www.sencha.com/products/extjs。

每年都会有很多新的框架发布，它们的作者并不是无聊想要新写一个框架，而是为了解决一个新的问题。比如 jQuery 的优势在于方便地操作 Web 界面（DOM）。而 Angular 并不是“更好的 jQuery”，而是提出一种新的解决问题的思路：通过数据绑定，让开发者直接修改数据模型，而不用关心界面（DOM）更新。GASP 库发现 jQuery 动画慢的问题，就重点优化 JavaScript 动画部分，它号称动画速度比 jQuery 快 20 倍，而且能开启硬件加速，在某些情况下比 CSS 动画性能还要好。

因此，不要迷信大框架，有时候越是有名的框架，越需要满足更多人的需求，会封装很多您可能不需要的资源进去。对于您来说，可能只需要一小部分功能，但是引入了一个庞大的库。我常常看到，在某些人的简单的个人博客中引入了一些庞然大物，但是其实没有必要。这对应聘者来说，是减分的。

在出现一些热门框架时，建议开发者先去了解框架的创建初衷，合理使用，而不是盲目收集。

岗位细分

我们知道前端的领域很广，所以有些大公司对它进行了更进一步的细分，

比如腾讯的两个职位“前端工程师”和“UI 工程师（UI Engineer）”。

UI 工程师 vs 前端工程师

在国外，UI 工程师是一个比较普及的岗位。以 Google 为例，一个叫 Front End Software Engineer，属于软件工程部，另一个叫 UX Engineer, Front End，有点类似 Front End 下的一个子项，属于用户体验设计部。

从使用语言的角度来分，UI 工程师只负责 HTML/CSS，前端工程师只负责 JavaScript，这是一种简化问题的解释方法。但我认为这两种职位的区分的重点并不是二者语言不一样，而是责任和关注点不一样。UI 工程师更关注视觉上和交互上的体验，而前端工程师更关注逻辑和数据方面的体验，二者是上下游合作的关系。

同时双方的工作也有一些交集，比如 UI 工程师也会负责一些交互或者动画相关的 JavaScript，前端工程师也要熟悉 HTML 标签才能用 JavaScript 去操作。双方都必须对对方的知识有足够的理解，合作才能进行下去。两种职位的目标是一致的：给用户提供更好的体验。

腾讯的 UI 工程师曾经叫“网页重构工程师”。这个名称来自一本很有名的床头技术杂文书《网站重构》（Designing with Web Standards），作者是世界上最有名的网站设计师之一，Zeldman,J.（泽尔德曼）。这本书的主要观点是，用 Web 标准来“重构”您的网站，而不要用以前的非标准的方式来做网站。

用一个我个人不太喜欢的大白话来说就是：不要用 table 标签布局，而要用 DIV+CSS[1]。我不喜欢这句话的原因是它不严谨，容易在纠正一个过往的错误的时候“用力过猛”。矫枉过正的后果是，现在有一些人只要看到 table 标签就觉得是非语义化的，喜欢用 DIV 搞定一切。但是 table 并不邪恶。table 用作数据表格的时候，是非常正确的。有些人在布局这一用途上用

1 简历中不要出现“DIV+CSS”这样的字眼，会减分；也不要出现 Dreamweaver，因为 Dreamweaver 是老式的“所见即所得”编辑器的代表。

DIV 干掉了 table，却开始对 DIV 上瘾。

2003 年《Designing with Web Standards》出版之前，全世界的设计师还没有 Web 标准的理念，都在用 table 标签布局，因为 table 可以让页面规整。这本书普及了 Web 标准的理念，在那之后，设计师开始使用语义化的 HTML 和灵活的 CSS 来设计页面。2005 年此书中文版出版后，也带来了全新的 Web 标准的理念。我第一次看这本书是 2009 年，那时我还在读大三，读完这本书之后几个月就签约到了腾讯 ISD 部门，职位是“网站重构工程师”，所以我对这本书有特别的感情。

不说远了，《网站重构》这本书的中文名是一个意译，表明在用 Web 标准来设计的过程中，我们要推倒以前的网站，“重构”一个新的网站。这也是“重构”这个词本来的意思——重新构造我们的代码。但是这个词被用在了一个希望能推进 Web 标准的职位上，给网站重构工程师这个职位带来了额外的风险：含糊不清。可能有人会认为这个职位一天到晚都在重写代码吧。

这本书的译者之一王宗义在知乎上说：

> “我是翻译《网站重构》一书的译者之一，当时我们 3 个译者各自起了不同的名字，这个名字是我起的，因为译者 3 个人中我是做程序开发的，借用了软件开发中的著名书籍《Refactoring》的中文翻译《重构》来影射当时国内网站需要用类似的方式来改变网站底层的本质。当时我们几个也有争议，不过阿捷和 dodo 最终接受了我的想法。就是没想到后来竟然能够成为 Web 前端的一个重要词汇。”

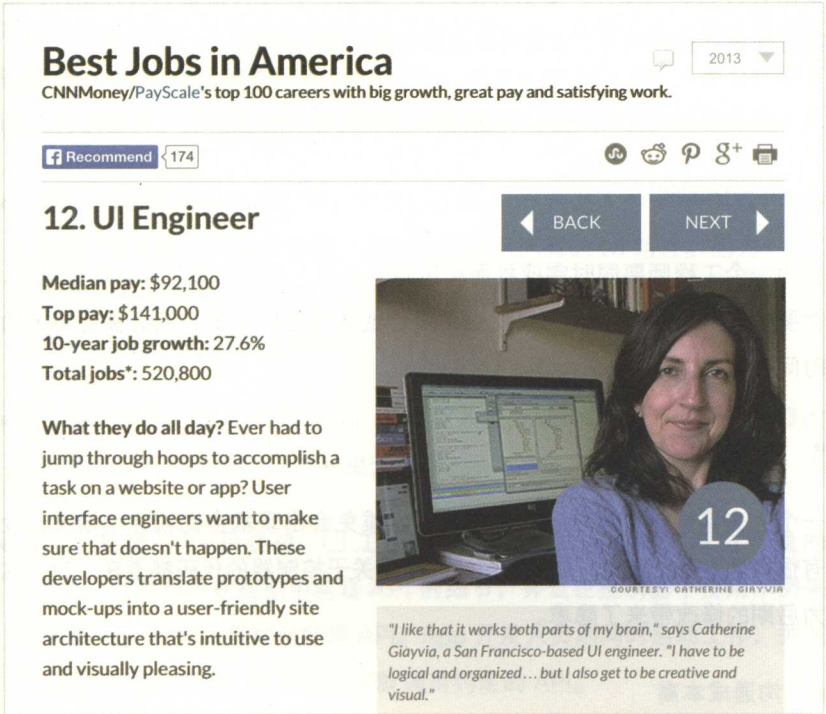
除了对岗位名字和职责的困惑，还有一个我常常被问到的问题是“重构只会 HTML 和 CSS，有什么前途？”

我的回答是，首先重构不应该只会 HTML 和 CSS。在前面“知识体系”一节中的所有知识点，重构工程师都需要了解和熟悉。特别是在性能、动画、SEO、可用性和移动等方面要有自己的优势。

不过，为了更好地跟国际接轨，同时避免“网页重构”与“代码重构”的

混淆，我们在 2015 年推动了职位更名的行动，现在我们已经正式更名为“UI 工程师”。

对于 UI 工程师来说，UI 开发的平台可能不会一直是浏览器，还有可能是原生 App[1]。



UI 工程师被 CNNMoney 评为 2013 年最佳工作，排名第 12，截图来自 money.cnn.com。

App UI 工程师

iOS 跟 Android 上的软件跟桌面软件，或者服务器端软件一样，都叫 Application。不过由于苹果 App Store 的普及，加上中国所有做移动端软件的团队的推广，现在大家都默认 App 就是指手机端上软件。本书遵循约定俗成的规范，提到 App 时，就是指智能手机上的软件。

1 大家都喜欢把它读成“诶批批”，就像一个缩写。但 App 不是一个缩写，而是对 Application 简写，所以正确地读法是 [æp]。

App 的市场在短短几年时间内就从无到有，它的发展速度比传统互联网要快得多。就像最开始的网站只需要一个开发者，而现在需要很多工程师协力合作，App 开发也在经历这样的变化。

传统的 iOS 开发流程是这样的：首先，设计师设计完 PSD 稿，做好标注，切出各种状态的图片，交给开发人员；其次，开发人员拿到各种切片，根据标注设计稿和切片，实现界面以及逻辑功能的开发。

从工程质量和进度角度讲，有这样几个问题。

- **开发周期长**

因为一个工程师要同时完成界面和逻辑的部分，所以二者只能按队列进行，需要较长的开发周期。如果发生了设计或者逻辑的变更，则会需要更多的时间去修改。

- **代码耦合强**

一个人去实现一个模块的时候，代码中难免会出现耦合比较强的情况，没有很好地 MVC 分离，关于视图的代码跟关于控制器的代码都混在一起，这为后期的修改带来了隐患。

- **沟通成本高**

因为设计师与开发人员之间通过标注和切片来沟通，但是标注本身就很不可靠，一个标注了所有间距的设计稿往往并不是我们需要的，工程师需要的是一些常量，以及当界面发生变化时的“规律”。

- **设计还原质量低**

传统的工程师在逻辑、健壮和成本上有非常敏锐的把控能力，但是在 UI 开发和用户体验方面的经验就略差一些。比如，标注了按钮与按钮之间的距离是 20px 并无太大参考性，因为按钮周围可能会有空白区域。如果开发人员迷信标注上的数字，在代码中直接写标注的数字，成品就会和设计稿效

果出现很大的偏差。而且由于设计师和开发人员之间沟通不畅、开发时间紧急和代码耦合的问题，导致设计还原的质量低。

在时间紧急的时候，工程师更注重性能问题和数据逻辑是否正确，而不太关注“按钮尺寸是否正确”这样的问题。

所以我希望推进的流程是从 Web 开发中借鉴经验，**让我们原本擅长用户体验的 Web UI 工程师来进行 App UI 开发**，而原来的 App 开发人员负责除了 UI 之外的部分。优化之后的整个流程大概是这样的。

- UI 工程师拿到需求单和设计稿之后，跟 App 开发人员一起沟通，明确哪些 UI 是这次需要重新做，哪些可以复用已有的 UI 组件。因为 UI 工程师可能刚接触到这个项目，需要清楚沟通，避免重复工作，也可以开始考虑如何建立公共 UI 组件。

- 如果是对于已有界面的修改，而无需改变逻辑的，UI 工程师直接修改界面代码和图片资源，然后提交测试。

- 对于新增的 UI 和逻辑，UI 工程师与 App 开发人员约定双方沟通的 API，然后自己在视图中实现 API 的细节，并且在控制器中使用示例来告诉 App 开发人员如何使用 API。App 开发人员则同时启动工作，关注后台和 App 逻辑，涉及视觉层就调用约定的 API。

- 界面和逻辑一起在测试环境上联调。

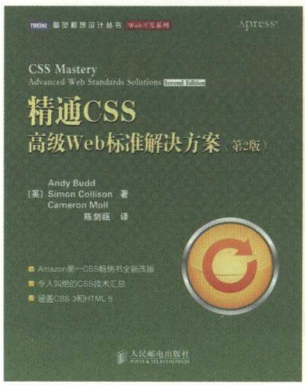
理想状态下，这个方案能解决上面的所有问题。不过有些同学可能会心里犯嘀咕，Android 原生 App 开发需要有 Java 的知识，iOS 开发需要熟悉 Objective-C 或者 Swift 语言，这些都不在前端工程师的技能树里面，如何能承担这部分的工作?

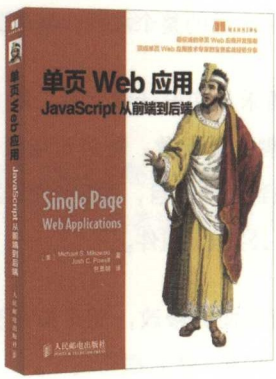
这就是我下一章要讲的：向移动端转型。

延伸阅读

1. 《精通 CSS: 高级 Web 标准解决方案（第 2 版）》（英）Andy Budd/Simon Collison/Cameron Moll，人民邮电出版社

2. 《单页 Web 应用：JavaScript 从前端到后端》（美）Michael S. Mikowski /Josh C. Powell，人民邮电出版社





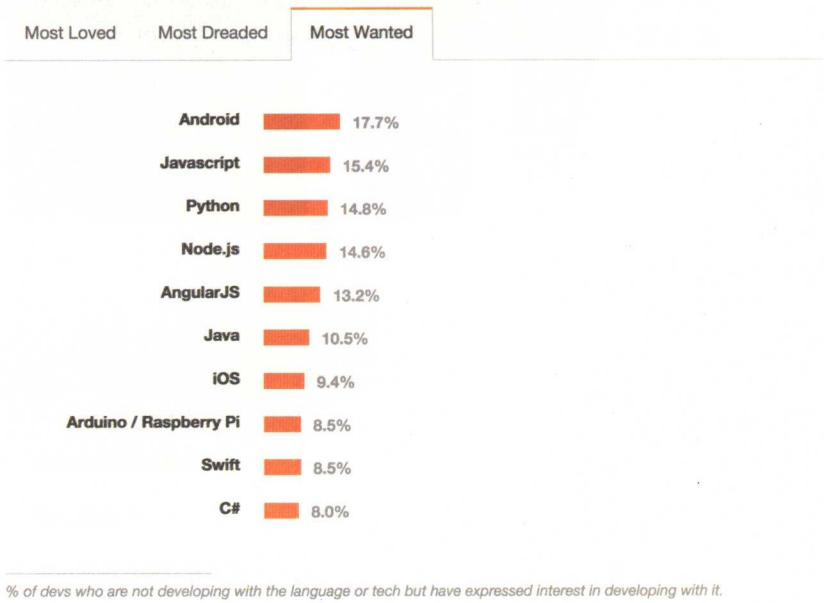
向移动端转型

早在一年之前，我们前端团队就开始思考如何向移动端转型这个话题。而到今天，向移动端转型已经变成越来越多的个人开发者和团队都开始思考和实践的事情。

为什么向移动端转型

技术是服务于市场的，在市场发生变化的时候，如果开发者不能顺应变化，就有被淘汰的风险，毕竟很多开发者所服务的这个岗位诞生都不到十年，消亡可能也会在十年之内发生。对于目标是全栈工程师的人来说，技术能力更是多多益善。

不光在中国，全世界范围的开发者们都有向移动端转型的趋势。根据 StackOverflow 的 2015 年开发者调查，对于“您最想学会哪种您现在还不会的技能？”这个问题，最多的回答是 Android 开发，第二是 JavaScript。



最想学的语言，截图来自 2015 年 StackOverflow 调查。

如今，对于独立开发者，面向移动端进行转型是非常不错的选择。因为移动端整体是一个上升很快的电梯，潜在用户人数爆发式增长，互联网巨头大力甚至赔钱推广用户支付习惯。

在很多年前，写独立软件的开发者如果想收费，只能用注册码的方式：用

户电汇转账给自己的账户，然后自己打电话过去告知注册码，成本非常高。

现在 iOS 绑定支付渠道，微信绑定信用卡，渠道跟支付能力已经绑定，搭上移动 App 的班车，海量有支付能力的用户就在眼前。只要 App 有市场需求，就会有用户买单。但是缺点也很明显，因为有越来越多开发者参与竞争。根据调查，近几年，独立移动端开发者的平均收入有逐渐降低的趋势。

对于在大公司供职的前端工程师，向移动端转型也是大势所趋。这里主要还是提到前端工程师，并不是因为我的职位是前端开发，而是因为在细分工种中，前端工程师是直接针对用户的客户端（在以前，这是浏览器）来编程的，当用户的客户端变成了手机浏览器和手机 App，前端工程师自然要转移战场。而对于后台开发者来说，技术还是基于 HTTP 等互联网基础设施，不直接接触用户，所以并没有太大的改变。

一个转型故事

谈到这个话题，我第一个想起的就是团队里一位同事，所以花一点笔墨来讲讲他的事吧。

我们团队是 UI 工程团队，两年前团队还没有普及 Mac，许多人还在使用 Windows 系统的 PC 办公。小 Y 自己买了一台 Macbook Pro 之后，开始利用工作之余的时间学习钻研 iOS 开发。

他为人比较低调，赚得“还不错”（据他所说），之后就辞职回家做全职 iOS 独立开发了。一年后，他的 Macbook Pro 也换成了 15 寸高配视网膜屏幕 Macbook Pro，他的策略是针对海外市场做工具类 App。选择海外市场是因为，当时海外用户已经形成付费的习惯，对于几美元的收费价格也习以为常；做工具类的原因一方面是兴趣所在，一方面不用运营内容，也不需要自己的服务器。

我很欣赏他，首先他是一个高效的开发者，同一时间开始学习做 iOS 的开发者还在翻阅书本，而那时他都上线一堆软件了。我曾经问他：“我有面

向对象编程的基础，也会 MySQL 数据库，这是不是做 iOS App 的一个优势？”他说：“有 Mac 就行。”我当时觉得这是一句玩笑话，现在觉得这是一句箴言——**行动重于计划**。

《重来》一书中有这样一句话：

> “如果您不想把手弄脏，就没法成为一个好外科医生。”[1]

因此，他坚信，自己做出的东西不仅能解决自己的问题，也能解决他人的问题。这也跟《重来》中的一句话是对应的：

> “自己给自己挠痒。”[2]

一定要是自己的产品的用户

风投在评估一个创业项目是否会成功的时候，有一个指标就是创始人是否是自己产品的目标用户。如果不是，那产品很有可能会失败。

国内某搜索引擎的搜索结果一直做不好，据我所知，他们的工程师都不用自己的产品。不知道这种恶性循环中，哪一个是因，哪一个是果？

此外，小 Y 做工具类的 App，并且也不怎么做宣传，而是让产品为自己代言。其实很多时候他在发布之前都不知道产品是否会受欢迎，因为市场是无法去猜测的，每一个点子都要靠一个“最小化可行产品[3]”去试探。卖得好，他就视情况去更新，卖得不好就算了。这里的“卖得好”可以理解成“用户有需求”，会热烈地使用和反馈，而不是说一开始就必须收费。

他的很多 App 的界面可以用“简陋”来形容，但是毫不影响销量和用户的赞扬。我们很多开发者想做一个 App 的第一步就是请一个产品和设计师来

---

1 You can't be a good surgeon if you don't get your hands dirty.
2 Scratch your own itch.
3 Minimum Viable Product，简称 MVP。

合作，组建一个专业团队。其实如果简化来处理，一些标准组件和开源控件就可以解决问题。

在这样的竞争下，小 Y 就显得很有优势。他可以快速测试，然后快速成功或快速失败。快速成功可以激发气势，增加动力。 哪怕小小的进步都能给您带来惊喜；快速失败可以让您快速掉头，不用纠结于“我已经花了半年的时间开发这个软件，我一定要将它发布”。一件事情耗得时间越久，您将完成的量就越少。在大公司，一些工程师士气低迷往往就是这个原因，成功来得很慢，失败也是。因为大家害怕失败，所以想把产品调整得完美无缺才发布。但是世界上成功的软件都不是完美的软件，而是在合适的时间发布的、刚刚够用的产品。如果它能活下来，在后面的版本中，它才有机会越来越好。

《精益创业》中有一句话：

> “客户需求只有在实际使用中才能辨明，再多的前期调研也只能发现客户认为他们想要什么，而不是客户实际上想要什么。因此在不了解客户真实需求的情况下，只会多做多错。”

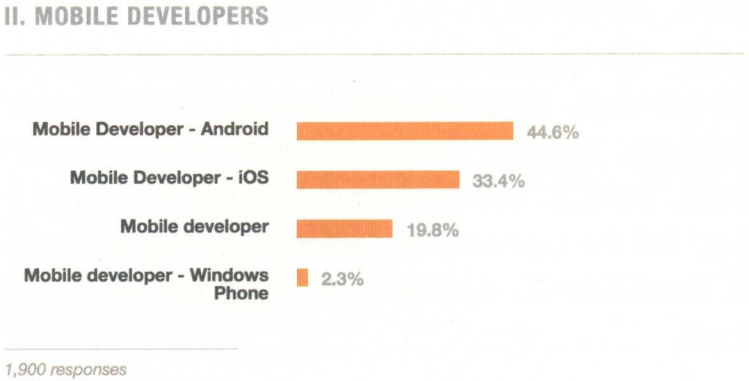
最后，小 Y 在市场中打磨了自己的手艺。脱口秀节目“罗辑思维”有一集是叫“夹缝中的 80 后”，里面说到 80 后的竞争力应该是“让市场认可您的能力，而不是让老板认可，因为老板会变，老板的标准不一致，而市场是一致的”。我不知道小 Y 是先听了这个节目，还是自己领悟出来，但他确实是这个理论的践行者。

有哪些方向

回到移动端转型的话题，往移动端开发转型一般是根据当前市场上的产品数量和自己的技术偏好来考虑。

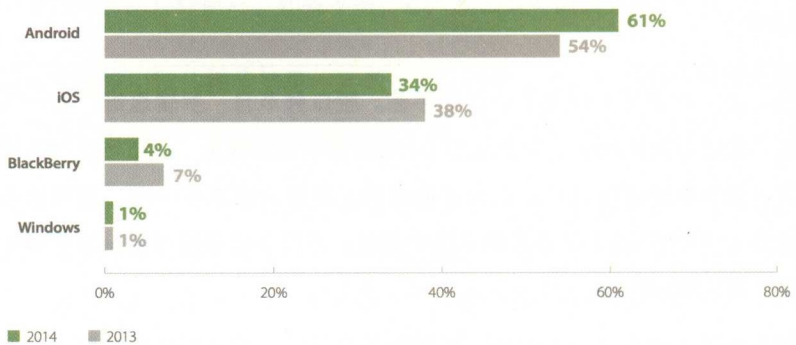
根据 2015 年 StackOverflow 的开发者调查，在总共 22000 份调查问卷中，

有 1900 名开发者表示自己主要针对移动端进行开发。其中，Android 与 iOS 开发者的比例是 4 ： 3，而只有 2% 的开发者专门为 Windows Phone 开发 App。此外还有 20% 的开发者并不限定为特定平台开发。



您针对什么平台开发？截图来自 2015 年 StackOverflow 调查。

这张图与操作系统的市场份额类似，它是 Millennial Media[1] 的数据。



各移动系统的市场份额，截图来自 2014 年 Millennial Media 调查报告。

从图上可以看出，Android 的市场份额正在进一步扩大，抢夺了部分 iOS 市场。

1 Millennial 是第一家上市的移动广告公司，由于智能手机及平板电脑销售量在全球的猛涨，移动广告市场飞速发展。

所以，让我先列出有哪些方向可以选择，希望我没有漏掉太多。

- iOS 原生 App

iOS 原生 App 开发的技能树相对比较新，需要学习 Objective-C 这门语言，以及 Xcode 的一些操作方法——主要是 storyboard，以及各种官方类库的使用方法。它带来的收益也很高，对于独立开发者，App Store 仍然是地球上最好的软件市场，对于团队，在未来 5 年都不会缺少对 iOS 开发者的需求。

- Android 原生 App

使用 Java 编程，如果有 Java 编程经验，Android 原生 App 是最好的选择，因为用户量和用户比例都在稳定增长。

- Windows Phone 原生 App

现在用户量还很少，除此之外也不知道如何评论……

- Web App

技术是最简单的，传统前端开发的技能树可以无缝移植，包括 HTML5/CSS3/JavaScript 等。应用场景包括浏览器中打开的 Web App、微信中的页面，或者混合模式 App。Web App 的好处是天然无缝移植到所有支持 Web 标准的平台——甚至 Kindle。

此外，对于中国开发者来说，微信公众号也是一个巨大的平台。之所以提到微信，是因为微信这个平台在中国的覆盖率几乎跟 Android 和 iOS 加在一起一样多，而且微信也有比较成熟的支付方式。

那么作为开发者，您该向哪些方向转移呢?

我觉得首先考虑的是自己的设备。您已经在使用 Mac、iPhone 或 iPad 了? 那么转型 iOS 原生 App 或者 Web App 是自然的选择；如果自己在用 Android 手机，那就为 Android 开发吧。

其次是对自己的学习能力和空闲时间有一个评估。还是在校学生？那给自己加点挑战吧，学习难度大一点的；如果平时工作比较忙，可以试试跟自己的知识相近的技术方案。

- 混合模式 App[1]

混合模式 App（Hybrid App）同时使用 Web 技术与原生程序语言开发，通过应用商店区分移动操作系统分发，需要用户安装使用。就像混合动力汽车使用汽油和电力两种动力一样，混合模式 App 使用两种技术制造。

混合模式 App 对于用户来说跟其他 App 一样，需要去苹果 App Store 或者 Android 应用商店下载。所以 App 需要对应的操作系统平台的技术，比如 Objective-C 或者 Java 制作整体框架。App 启动后，它的全部界面或者部分界面中，使用网络视图（WebView）技术来实现。WebView 能加载显示网页，可以将其视为一个浏览器，它一般使用 WebKit 渲染引擎加载显示网页。

如果对 WebView 进行一些性能优化和设计体验的优化，用户可能感受不到当前界面是用 WebView 还是用原生技术制作的。一些常用的优化方法如下所述。

- 把 WebView 的部分或者所有资源打包在 App 中

需要网络数据时，可以通过网络请求 json 或者体积比较小的数据格式，然后通过本地页面模板和资源来渲染。这种方法的缺点是，App 发布包体积会变大。

- 把需要加载的资源设置好预先加载

可以在 App 启动时从后台下载需要的资源，并缓存在手机沙盒中备用。这种方法的好处是不会增加包体积，不过第一次访问的时候可能因为没有预

1 混合模式 App（Hybrid App），即原生技术与 Web 技术相结合的一种 App，兼具原生 App 良好用户交互体验的优势和 Web App 跨平台开发的优势。

加载资源而导致等待时间比较久。

- 使用 HTML5 Manifest 技术实现资源缓存

HTML5 引入了应用程序缓存，这意味着 Web App 可进行缓存，并可在没有互联网连接时访问。这种方法的好处是，缓存所有资源到本地之后，如果希望更新 WebView，可以在服务器上更新资源列表和 Manifest 文件。App 检测到 Manifest 文件的修改，就知道资源已经更新，可以开始下载新的资源了。

- 不要把整个 App 的主要逻辑都使用 WebView 来实现

要结合原生技术和 WebView 各自的优缺点，根据不同的场景选择合适的技术。原生技术的优点在于能很好地操作 App 存储数据；实现页面间切换、高性能动画、大量数据的界面（比如可以无限滚动的图片流）。WebView 的优点在于开发快、技术简单；前端开发者能够利用已有的 CSS3 和 JavaScript 知识；页面能够从服务器端更新；能够分享到社交平台；在多个平台上共用等。

- 设计得更像一个 App，而不是一个网页

在这种做法诞生初期，我们还会把 App 分为原生 App 和混合模式 App，不过从 2014 年以来，我们已经不会这样区分了。现在我们一般认为，一个 App 中有一些 HTML 页面是非常自然的事情，所以这个概念在渐渐淡化。

WebView 与原生代码通信

WebView 中的逻辑是由 JavaScript 来实现的，而 WebView 之外的逻辑是由原生代码来实现的，二者之间如何实现通信呢？一个常见的例子是，点击 WebView 中的一个按钮，我希望弹出一个原生的警告框，应该怎么实现呢。

在 Android 中，可以使用 `WebView.addJavascriptInterface` 方法

来实现互相通信。WebView 的 addJavascriptInterface 方法有两个参数，第一个参数就是我们一般会实现一个自己定义的类，类里面有要提供给 JavaScript 访问的方法；第二个参数是一个 JavaScript 对象名，它在 WebView 中作为刚才我们自定义的类的一个实例。

在原生代码中定义好类和对象之后，就可以在 WebView 中通过 JavaScript 调用原生代码了。调用模式为 window．对象名．方法名 ()，或者是 javascript：对象名．方法名 ()。

在 iOS 中，原生代码没有这样简便的方法，想要接收 WebView 中的函数调用时，需要使用 WebView 的 shouldStartLoadWithRequest 委托。

首先，在希望发起通信的 WebView 中设置这样一个自定义协议的链接： 一个按钮 这是一个非常简单的链接，不过链接的协议是我自定义的 myapp，也可以是任何其他与 App 相关的名字。

然后在调用该 WebView 的原生代码中，添加对该 WebView 的委托监听：

```
- (BOOL)WebView: (UIWebView *)WebView shouldStartLoadWithRequest:
(NSURLRequest *)request navigationType:(UIWebViewNavigationType)
navigationType{
   NSURL *url = [request URL];
   if ( [[url scheme] isEqualToString:@"myapp"] )
   {
       //NSString *slug = [url path];
       //slug 就是刚才的 somepagename
      // 得到了这个信息，我们可以做一些原生的操作了
   }
   return YES;
}
```

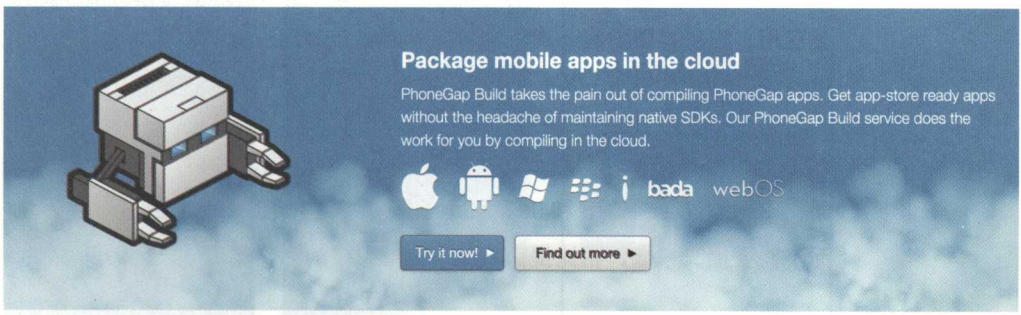
发生了什么？首先我们自定义了一个协议名称 myapp，然后在 WebView 中发起了一个以该协议开头的请求 myapp://somepagename。WebView 发起任何请求之前都会通过外部 shouldStartLoadWithRequest 的“过滤”。所以在这里的函数体中，我们增加了对协议的判断，如果是我们自定

义的协议，就不要跳转过去了，直接执行自己定义的逻辑即可。

从原生代码向 WebView 传输数据的原理比反过来要简单一些，这里就不再赘述了。

混合模式 App 开发框架

从上面这个简单示例种可以看到，使用 WebView 和原生技术相结合的方案还是比较复杂的。而且要考虑到 iOS 平台和 Android 平台的差异性，这可能会让前端工程师感到抓狂。好在，PhoneGap 这样的跨平台开发框架应运而生，它让开发者只需要使用 Web 技术就可以创建跨平台的混合型 App。



PhoneGap 可以让开发者使用 Web 技术创建跨平台的混合型 App，截图来自 phonegap.com。

PhoneGap 通过对各个平台底层功能进行封装和抽象，然后通过 JavaScript 暴露出一致的 API，让开发者可以通过 JavaScript 编写跨平台的原生 APP。

虽然看上去“一次编写、到处运行”的愿景很美，但是 PhoneGap 有这样几个缺点。

- PhoneGap 的编程语言其实是 JavaScript，这对于非前端工作者来说，学习起来和学习原生的 Objective-C 或 Java 编程语言难度差不多，想精通 JavaScript，相当不易。
- PhoneGap 编译的 App 包大小比一般的会大很多。

- PhoneGap 的目标是方便地创建跨平台应用，但是苹果和 Google 都发布了自己的人机交互指南。有些情况下，iOS 程序和 Android 程序有着不同的交互原则。使用 PhoneGap 就意味着您的程序在 UI 和交互上，既不像原生 iOS 程序，又不像原生 Android 程序。

- 动画性能不佳。JavaScript 终究无法和原生程序比运行效率，当制作一些动画效果，或者有大量数据的长页面的时候，就表现得很明显。

当然，PhoneGap 的优势也很明显。

如果是做图书类、查询类、小工具类应用的话，这些应用 UI 交互不复杂，也不占用很高的 CPU 资源，PhoneGap 将很好地发挥出它的优势。对于这类应用，您只需要编写一次，则可以同时完成 iOS、Android 和 Windows Phone 等版本的开发。

如果希望版本更新，而且改动不大，只是内容升级，那它升级时只需要更新相应的 HTML、CSS 和 JavaScript 文件。这些资源都可以通过云端更新，而不需要提交审核，相比提交 App Store 审核的话，能节省一周时间。

所以 PhoneGap 不是万能的，但也不是没有用，它有它擅长的领域，一切都看您是否合理地使用它。

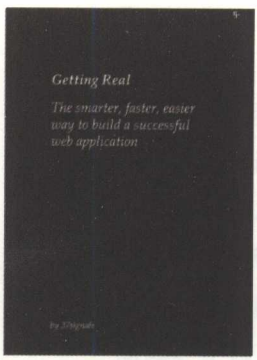
如果希望更好地开发混合模式 App，最终还是需要学习对应平台原生的开发技能，以及人机交互指南。

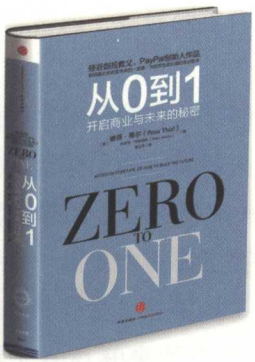
延伸阅读

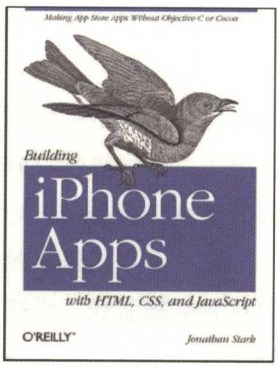
1. 《Getting Real》（美）Jason Fried / Heinemeier David Hansson / Matthew Linderman，37signals

2. 《从 0 到 1：开启商业与未来的秘密》（美）彼得·蒂尔、布莱克·马斯特斯，中信出版社

3. 《Building iPhone Apps with HTML, CSS, and JavaScript》（美）Jonathan Stark，O'Reilly Media







持续集成

持续集成（Continuous integration，CI），是一种软件工程流程，将所有工程师对于软件的工作副本，每天整合数次到共用主线上。

只有一个工程师维护的项目，不需要持续集成，他可以随时提交自己的修改，无须担心与其他人的代码产生冲突。但随着软件项目复杂度的增加（即使增加一个人），就会对集成和确保软件组件能够在一起工作提出了更多的要求。

无论什么样的软件开发，只要是多人开发，就面临持续集成的问题。不过对于 App 和桌面软件这样的现代软件，它们有自己的集成开发环境，挑战稍微小一些，只需要考虑版本控制和代码合并即可。而对于服务器开发和前端开发，它们没有固定的集成开发环境，所以在构建持续集成的开发流程中，就要考虑版本控制、包管理、依赖关系、架构优化、自动化发布等一系列问题。

版本控制

包管理

构建工具

版本控制

我们首先从版本控制开始。版本控制在编程项目中的重要性常常被低估和误解。

版本控制，就是利用版本控制工具（Version Control System，VCS）来管理项目从开始到定案的过程。

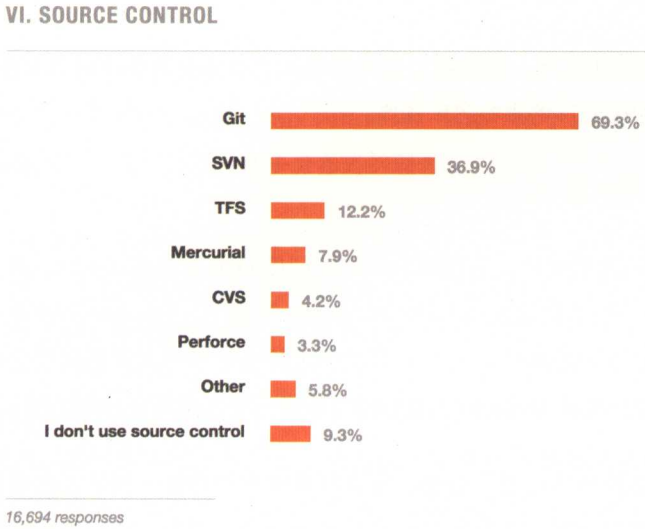
为什么说被低估了，这是因为大学教育和公司的入职教育都普遍认为版本控制系统的使用是非常简单的：拉取代码，修改代码，提交代码，推送代码。操作过程确实是非常简单，但是背后的原理以及分支类的操作往往就没有深入介绍。

另外，很多工程师常见的误解是，版本控制系统是帮助他人（老板或者项目经理）追溯进度，或者帮助团队代码合并的工具，而不知道用好版本控制系统最大的受益者是自己。版本控制系统是帮助自己解放大脑的工具。由于这种错误的理念，这些工程师会在本地留下大量“脏代码”。我在不少新人的本地代码库里看到过这种情况。

脏代码的产生可能是由于某个烂尾项目，或者某个自动化工具的临时输出，或者第三方依赖中进行的参数修改。而用版本控制系统提交代码的时候，可能需要花上半分钟的时间甄别哪些文件需要添加到本次提交，哪些需要剔除。

错误地把版本控制工具理解成协作工具，或者发布工具的话，就会等到某个功能正式完成之后再提交，这样当出现问题的时候很难回溯。

根据 2015 年 StackOverflow 公司进行的开发者调查，使用最广泛的两种版本控制工具是 Git 和 SVN，而且有趣的是，直到现在还有 10% 的开发者不使用任何版本控制工具。所以我想谈谈 SVN 和 Git 这两种流行的版本控制工具的设计异同，以及在项目中的应用。



您使用什么版本管理工具？截图来自 StackOverflow 官方调查结果。

SVN

Apache Subversion（SVN）是在企业中应用很广泛的版本控制工具，它最初的开发目的是为了替代老牌的 CVS（Concurrent Versions System），CVS 有以下几个缺点。

- CVS 不支持文件重命名，如果您重命名一个文件的话，之前的修订历史记录就会丢失。
- CVS 没有原子性提交，如果您提交很多文件，提交到一半的时候出错，那就有点麻烦了。
- CVS 只支持文本文件，无法提交二进制文件。

SVN 解决了这几个问题，并且加入了一些新的特性。

- SVN 对二进制文件的版本管理，使用了节省空间的保存方法，只保存

和上一版本的不同之处。

- 目录也有版本历史。重命名、复制和删除也会保存在版本历史记录中，当然，要用 SVN 的重命名来操作。
- 分支的开销比较小。

由于 SVN 的这几项优化，它已经渐渐替代 CVS 成为主流版本控制工具（至少在 GitHub 流行之前是主流）。在 Windows 系统下，TortoiseSVN 是一款非常受欢迎的客户端软件，它的主要成功之处在于与资源管理器整合得相当不错，可以在文件或者文件夹上右键单击进行绝大多数 SVN 操作。

集中式代码管理的核心是服务器，所有开发者在开始新一天的工作之前必须从服务器获取代码，然后开发，最后解决冲突，提交。所有的版本信息都放在服务器上。如果脱离了服务器，开发者基本上可以说是无法工作的。一般的 SVN 开发者的一天是这样的。

- 早上到达公司，从服务器拉取项目最新代码到本地。
- 切换到自己的分支，开始工作，每隔一个小时向服务器自己的分支推送一次代码。
- 下班时间快到了，把自己的分支合并到服务器主分支上，一天的工作完成，并反映给服务器。

如果不能连接到服务器，基本上无法工作。看上面第二步，如果服务器不能连接上，就不能提交、还原、对比，等等。而且如果太多人同时操作，服务器就会面临过大的压力，也可能处理过慢或者崩溃。

在企业内部，使用 SVN 没有什么问题，服务器压力和内部带宽都能够承受所有员工一起操作 SVN。但是在开源世界，这种架构方法就不行了，著名的开源软件的开发人数太多了，因此诞生了 Git。

Git

Git 是一个分布式版本控制软件，是天才工程师、Linux 内核开发者 Linus 开发，目的是更好地管理 Linux 内核源码。其第一版于 2005 年发布，它与 SVN 最大的不同之处就是基于分布式的理念。

SVN 需要有一台中央服务器，所有的分支（branch）、主干（trunk）和标签（tag）全都保存在这台中央服务器上。开发者需要提交代码时，需要保持跟中央服务器的网络连接。切换分支时会有服务器与本机的数据交互。

Git 改进了这一点，在每台安装有 Git 的机器上都有所有的版本和历史记录，所以可以直接在本地切换分支。这带来的好处是创建和切换分支非常快，而且工程师在没有网络的条件下也能够提交和管理自己的版本，等到有网络时再同步到公共库。

那么自己本地机器上的 Git 库，如何与其他人的代码库合并呢？答案是，每个 Git 库都可以设定一些远程库（remote）地址。remote 是远程服务器的意思，表明本地代码库跟远程服务器代码库的一个对应关系。默认如果从 GitHub 上克隆（clone）一个代码库下来的话，会有一个名为 origin 的 remote。比如切换到一个 Git 目录，运行 Git remote -v，会显示：

```
$ Git remote -v
origin https://GitHub.com/yuguo/yuguo.GitHub.io (fetch)
origin https://GitHub.com/yuguo/yuguo.GitHub.io (push)
```

origin 是起源的意思，表明我的代码库是从这个源克隆下来的，随后也可以把修改后的代码推送到这个源。

使用 Git 部署代码

一个 Git 代码库可以配置很多远程库，除了 remote 这个默认远程库，还可以新增其他的远程库，比如新增一个叫做 prod 的远程库，地址设置为生产环境的地址。

```
$ Git remote add prod your-ssh-username@your-host:/var/www/yoursite/.Git
```

然后在自己的服务器上开启 SSH 支持，安装 Git，并且设置一个接受 Git post 的钩子就成了。

```
cd /var/www/yoursite
Git init .
Git config receive.denyCurrentBranch ignore
Git config --bool receive.denyNonFastForwards false
cd .Git/hooks
wget http://yuguo.us/post-update
chmod +x post-update
```

现在我们就可以把源码直接推送到生产环境中去，直接上线。

```
$ Git push prod master
```

这句代码会把本地的主干代码（master）直接推送到服务器（prod）上并生效。

看上去比较复杂，不过只需要配置一次，后续就很方便了。对了，我用 SVN 时习惯用 GUI 软件 TortoiseSVN，但是用 Git 时更喜欢用命令行的界面来操作。

在公司工作时，必须遵循规范使用 SVN。在制作个人项目、小玩意、插件开发、个人博客、团队的开源项目时……我更倾向用 Git。Git 命令行在跨平台上有很大的优势，熟悉了 Git 命令行操作之后，分支和提交操作都非常快，而且在 Windows 和 OSX 下都可以运行。

版本控制最佳实践

下面我摘录了一些 StackOverflow 上面关于 SVN 的最佳实践，其中有一些操作同样适用于 Git。

- **鼓励频繁地提交**

SVN 的初学者可能会有一种想法，他们保留代码一直在本地修改，直到代码确定没有问题了才提交。但最佳实践是频繁地提交，而不要等到代码没问题了再一次性提交。

对于可能损坏主干原则的代码，不要直接提交到主干，而是创建一个分支，

在分支中频繁提交。

- **确定分支流程**

基本上所有的特性和较大的 bug 修复都应该使用分支来修改。

- **定义主干原则，并且坚守它**

我们团队的主干原则是“主干对应的代码必须是可以发布并且不会产生 bug 的”，如果不能保证新增的或者修改的代码符合这一原则，就在分支提交代码。任何人破坏这一原则引起 bug，就请大家吃饭。

- **不要把逻辑的修改和代码格式化操作混在一起**

如果您做了一些代码格式化的操作，就单独提交这次修改。比如您去掉了代码中所有的空行，那就单独提交一个 commit，然后再做一些逻辑的修改，再提交。这样可以避免“天哪，所有的东西都不一样了”，出现问题之后更容易追溯。

- **不相干的代码分开提交**

也就是说不要在一次提交里修复两个 bug。

- **保持工作代码库的“干净”**

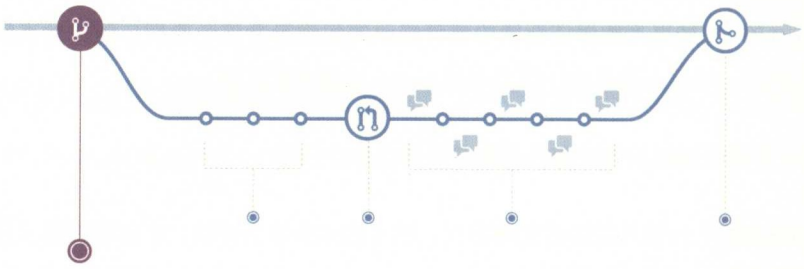
如果您有文件不想也不需要提交，就加入到忽略列表（ignorelist）。不需要提交的文件包括编译后文件、配置文件和第三方依赖等。这样的好处是，您每次打开 SVN 提交界面，如果没有修改过任何代码，就会看见一个空的列表，如果修改过代码，就显示修改过的代码。这能提醒您不要漏掉任何需要提交的文件。

GitHub 工作流

GitHub 是全球最有名的 Git 源码存取服务提供商，也成为了工程师们思维

碰撞的社区，一些非常优秀的开源作品正是在 GitHub 平台上由很多互不相识的工程师协作创建。为了达到这一目标，GitHub 鼓励采用一套标准的工作流（GitHub Flow）来创建项目，这样来自世界各地的工程师也能采用标准化的流程来协作。更理想的情况是，把 GitHub 工作流推广开来作为业界标准，即使您离开了一家公司，下一家公司也采用这样的工作流。

下面我介绍一下 GitHub 工作流。



GitHub 工作流，截图来自 GitHub 网站。

- 创建一个分支。在 Git 中，主分支叫做 master，不要直接在 master 中修改代码。GitHub 工作流在开发任何新特性时都会创建一个分支，这相当于一个新的工作环境。

- 在这个分支中提交您的代码。您可以在里面随意提交——事实上它鼓励您频繁地提交。并且建议每一次提交都清楚地写上原因，因为这是您的“脚印”。

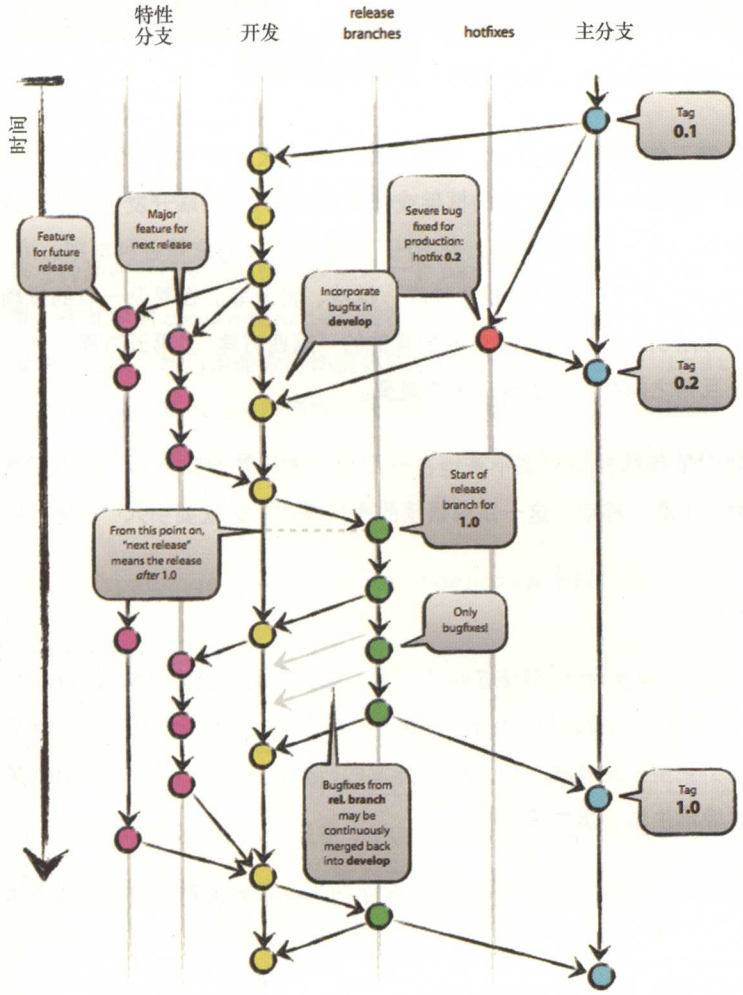
- 当您觉得您的某个任务完成了，发起一个推送请求（Pull Request）。推送请求是公开透明的，所有人都会看到您的代码跟当前代码的不同，也知道如果接受了您的请求会发生什么。如果您想给一个开源项目贡献代码，那么在发起推送请求的时候，项目负责人会看到您的修改。如果您是作为合作者修改项目，发起推送请求也会让代码审查者来查看您的代码，在一个推送请求的页面下面，大家可以进行讨论。GitHub 工作流与 Git 工作流有所不同的就是它非常鼓励在推送请求中讨论。

- 大家检查并讨论您的代码。如果大家反馈您的代码有 bug，或者不符合

项目编程风格，您可以在您的代码库中继续提交，GitHub 会显示您的新提交，大家可以根据您的新提交，进行新的讨论。

- 合并和发布。如果您的代码通过了审查和测试，这时候就应该合并到主分支了。如果您觉得需要在合并到主分支之前测试合并之后的效果，可以在本地进行测试。

与 GitHub 工作流相对应的还有 Git 工作流，但是 Git 工作流比较复杂，如下图。相比而言，GitHub 工作流更加简单，适合小团队来快速上手和迭代。



Git 工作流，图片来自 nvie.com。

包管理

在谈包管理这个话题之前，首先要了解什么是“依赖关系”。

让我们从历史悠久的 Linux 哲学开始。Linux 工程师都是哲学家，他们编程有很多原则，其中有一句金句，可以应用到几乎所有语言的编程：

> 一个程序只做一件事，并做好。

这个哲学也可以扩展到为人处事的方式，GTD（Getting Things Done）原则建议您对一个大的项目进行拆分，分为很多小的可执行和评估的子任务，这样每一件小事情都可以做得很好，效率也会提高。

在我们日常工作编写的软件中，可能有绝大部分代码都不是我们自己输入的。我们“依赖”一些第三方的框架或者库。在 Web 前端开发中，我们依赖各种框架、库、静态资源等；在 PHP 开发中，我们依赖各种框架、库；在 iOS App 开发中，我们依赖各种库、模块、资源等。在复杂一点的依赖环境中，您所引入的第三方库也依赖其他的“第四方库”“第五方库”……如何保证互相之间都不会出现冲突很重要。

如何让我们依赖的资源有条不紊地在一个地方进行管理和更新，而不用重复“搜索、下载、移动”这一系列繁琐的手工操作？这就要引入“包管理”。

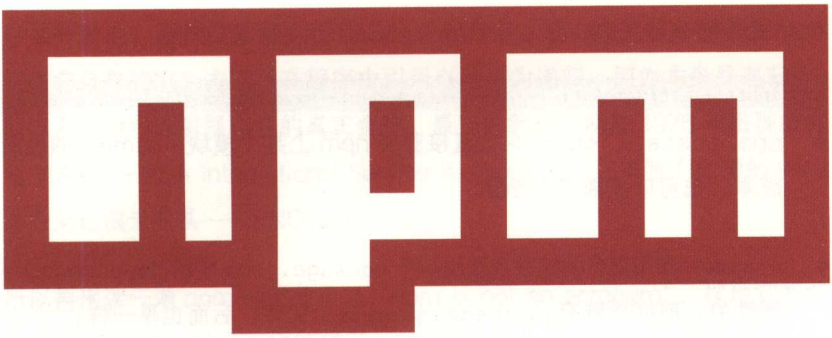
Node.js

Node.js 的包管理器 npm[1] 应该是世界上最有名的包管理器。如果没有 npm，Node 不会有今天的普及度。npm 收集了大量优秀的 Node.js 代码包，然后这些库吸引更多开发者进入 Node.js 开发的行列，反过来又促成了 npm 的繁荣，就像鸡生蛋，蛋生鸡一样。

npm 有一个公共的组件库，全球所有的 Node.js 开发者都可以发布或者下

1 Node 包管理器，http://www.npmjs.org/。

载公共组件。npm 可以通过命令行或者 package.json 来下载所依赖的组件，并使用自己的缓存机制优化下载效率，安装所有组件，甚至包括一些包含 C++ 代码的组件也可以安装。



npm，图片来自 npmjs.org。

npm 引入依赖组件的方法如下所述。

第一种是在自己项目的根目录里写一个 `package.json`。这是一个 json 对象，在其中的 `dependencies` 或者 `devDependencies` 值列出所需要的模块和版本，然后用命令行切换到项目根目录，运行 `npm install`。通过这种方法，其他人在得到您的代码之后，仅需要一个 `package.json` 文件，就可以简单地使用 `npm install` 命令来安装所需要的所有依赖。模块会全部下载到 `node_modules` 文件夹。

这是一个 `package.json` 中的 `dependencies` 字段代码示例。

```
"dependencies": {
    "gulp-util": "~2.2.14",
    "through2": "~0.4.1"
},
```

因为 `node_modules` 文件夹里全都是第三方代码，实际上是脱离于自己项目的代码库的。所以应该在 `.gitignore` 或者 SVN 的忽略文件列表里忽略掉 `node_modules` 整个文件夹，而且所有项目成员也不应该修改 `node_modules` 里的任何东西，否则在将来 npm 安装的时候可能会丢失

您的修改。如果发现某个模块有修改的必要，要向原作者提出 issue[1] 或者推送请求。

除了手动编辑 package.json 来维护整个项目的依赖包之外，还有一些 npm 命令会直接修改这个 package.json，比如下面这句。

```
npm install &lt;name&gt; [--save|--save-dev|--save-optional] [--save-exact]
```

- npm install <name> 会直接安装 npm 上某个模块（name）的最新版本，也可以选择带上参数。
- --save 表示这个模块会直接出现在 package.json 中的 dependencies 属性中。前提是要有 package.json 这个文件，后面也是一样。
- --save-dev 表示这个模块会直接出现在 package.json 中的 devDependencies 属性中。关于 dependencies 和 devDependencies 的区别，假设一个人下载了您的模块并且在代码中使用，他只需要您的功能，并不需要您的测试代码，或者文档构建器。这时候最好把这些开发阶段使用的包放在 devDependencies 中。
- --save-optional 表示这个模块会直接出现在 package.json 中的 optionalDencies 属性中。如果某个模块是可选的——这意味这您在您的依赖代码中做了备选方案（fallback）处理——就放在 optionalDependencies 中，npm 安装的时候会尽量下载和安装这个模块，但是如果下载失败了，npm 也不会报错，仅仅跳过去。

此外还有一个参数 -g，意思是将模块安装在全局目录。Linux 的全局目录默认是 /usr/local/ 或者安装 node 的地方，而本地安装是当前项目的 node_modules 目录。在本地安装的包一般使用 require() 来把模块引入到代码中，而使用全局安装的模块可以在命令行中直接使用。

关于 npm 这个词的意义，还有一个有趣的典故。一般人可能以为是 Node

1 issue 是提出问题的意思。

Package Manager 的缩写，但实际上不是这样的。作者声称，npm 不是 Node Package Manager 的首字母缩写[1]，所以不能全大写。npm 是“npm is not an acronym”（npm 不是一个缩写）这个递归定义的简写。

这里有点绕，必须先解释一下什么是递归定义。递归定义是一种在定义中引用它自身的定义方法，在程序中引用自己称为递归。因为本身是递归的，所以无法精确地解释出它的真正全称。最早的例子可能是 1976 年出现的缩写 VISA——VISA International Service Association。还有我们熟悉的 PNG 的定义也很无厘头——PNG is Not Gif。

所以再来读一遍 npm 的递归定义“npm is not an acronym”，按理它的缩写应该是 ninaa，但是这样就成了 acronym（缩写）了，就是错误的。最终，我们只能简称它为 npm，我们不知道它是什么，但可以肯定的是，它不是一个缩写。

说到底，作者就是想把我们所有人绕进去。下面是一段让人抓狂的对话。

> npm 可以大写吗？
>
> 不可以，因为 npm 不是一个缩写呀！
>
> 那 npm 是什么单词的简写吗？
>
> 是“npm 不是一个缩写”的简写。
>
> ...

Bower

除了 npm，各种语言也有对应的包管理器。我想第二个隆重介绍的是 Bower，第一个原因是因为它是包管理器中的新人，也是备受瞩目的 Yeoman[2] 三剑客中的一个。第二个原因是 Bower 管理各种前端资源，这对

1 在这里，“缩写”指首字母缩写，比如 Information Technology 缩写成 IT。“简写”指取一个长单词中的一部分，比如 Application 简写成 App。

2 Yeoman 是 Google 团队开发的一套开源前端工作流套件。它包括 3 个工具：Yo（脚手架工具）、Grunt（构建工具）和 Bower（包管理器）。

于我本身的工作是非常需要的。第三是前端的依赖模式跟普通的后端语言有所不同。

比如制作一个网站，可能需要最新版的 jQuery，简单的方法就是从浏览器中搜索 jQuery，然后找到 jQuery 官网，将压缩版本下载到本地。我们把压缩代码文件复制到项目的 js 文件夹，然后在页面中引入这个 jQuery 文件的路径，就可以使用 $ 这个变量了！

这种手工维护方式会导致项目得不到 jQuery 更新带来的好处。类似的前端依赖还有很多，比如十来个 jQuery 插件，AngularJS 或者 Bootstrap 等 CSS 库……维护所有这些前端包是非常让人苦恼的。

Bower 就是为了解决这种问题而诞生的，在自己的项目根目录设定一个 Bowerfile.js，然后写上自己项目的依赖，就可以一句命令更新所有的组件。前端使用场景中，为了最大化利用用户缓存，少传输数据，所以很多资源都是分批加载的，特别是第三方的资源，而不会一股脑全合并在一起。所以 Bower 没有系统级的依赖，在不同 App 之间也不互相依赖，依赖树是扁平的。

Bower，图片来自 bower.io。

举个例子，Bootstrap 对 jQuery 有依赖，所以在 Bootstrap 的 Bower.json 中有以下代码。

```
"dependencies": {    "jquery": "&gt;= 1.9.0"  }
```

这样，在运行代码 bower install bootstrap 之后，命令行会提示：

```
bootstrap#3.1.1 bower_components\bootstrap
└── jquery#2.1.0
```

但是实际上，jQuery 会安装在 Bootstrap 的同一层级目录。

因为所有的资源都是扁平的，所以如果页面上有依赖关系，比如 js 依赖，就要自己把每个单独的库引入之后，使用前端方法引用它们——比如 seaJS——这就不是本文讨论的话题了。

Bower 比起 npm 有一个有意思的特性，它并没有架设自己的服务器来托管所有的包，而是直接运行在 Git 之上。npm 发布一个包会发布到 registry.npmjs.org 或者对应的镜像上，但是 Bower 包发布是将一个包名绑定到一个 Git 仓库，然后代码本身推送到 Git 仓库即可。

```
$ Bower register mything Git://GitHub.com/username/mything.Git
```

这样的话，mything 这个名字会注册到 Bower 库。

跟 npm 一样，Bower 除了可以依据配置文件来下载依赖文件到本地，还可以通过命令行直接下载。Bower 提供了几种方法来下载资源。

对于注册了的资源，直接使用注册名，比如 jQuery。

```
$ Bower install jquery
```

GitHub 用户名 / 项目名的简写

```
$ Bower install desandro/masonry
```

Git 终端

```
$ Bower install Git://GitHub.com/user/package.Git
```

URL

```
$ Bower install http://example.com/script.js
```

其他软件包管理器

几乎所有的现代语言都有自己的包管理器。

Composer 是 PHP 中的包管理工具，它可以让您声明自己项目所依赖的库，然后它将会在项目中为您安装这些库。

Ruby 的包管理器是 gem，它使用一个 Gemfile 来声明依赖的包。

CocoaPods 是一个应用程序级别的包管理器，它能够管理 Objective-C 的包或者任何基于 Objective-C 环境来运行的语言包。它使用 Podfile 来声明依赖的包。

关于版本号

无论是什么语言，每个包发布之后都不是一成不变的，代码总是有更新：增加功能、修复 bug 等。包发布的版本号需要遵循一定规范。

根据 semver 的规范，版本号用小数点分隔为三个数字。比如 v3.2.1 中 3 是主要版本号，2 是次要版本号，1 是补丁。

- 主要版本号：有 API 变更导致不兼容旧版本的时候使用。
- 次要版本号：新增功能，但是向前兼容的情况下使用。
- 补丁：修复向前兼容的 bug 时使用。

在 npm 的 package.json 中可以用一些语法指定需要某个模块的哪个版本。

- 1.2.3, =1.2.3：指定版本为 1.2.3。
- >1.2.3, <1.2.3：大于 / 小于 1.2.3。
- >=1.2.3, <=1.2.3：大于等于 / 小于等于 1.2.3。

- 1.2.3 - 2.3.4：大于 1.2.3 并且小于 2.3.4。
- ~1.2.3：合理地靠近 1.2.3，等价于 >=1.2.3-0 <1.3.0-0，1.3.0-beta 不满足这个判断条件。
- ~1.2：等价于 >=1.2.0-0 <1.3.0-0，所有以 1.2 开头的版本，同样等价于 1.2.x。
- ~1：等价于 >=1.0.0-0 <2.0.0-0，所有以 1 开头的版本，等价于 1.x。
- *：任意版本。

要根据项目的具体情况来看如何使用模块的版本，如果限制得太严格会需要频繁关注版本的更新，然后修改自己的 package.json。如果限制得太松散，可能会有新版本带来的 bug。一般使用 ~1.2.3 就好。

使用依赖管理工具可以让您的大脑从版本管理的琐事中解脱出来，从而大大提高工程师的效率。

构建工具

前端开发与 App 开发有很多不同，不过从开发环境来讲，最大的区别是前端没有一个固定的集成开发环境，也不存在源代码和结果码的区别。前端所有的代码、资源都是开放透明的，HTML、CSS、JavaScript 代码下载到浏览器，就是纯文本，这里没有任何秘密。

所有的脚本语言都是直接运行，不需要编译成可执行代码，只不过 Python 等脚本运行在服务器，用户看不到源代码，而 JavaScript 运行在浏览器，所有人都能看到源码[1]。有时候出于保密的考虑，会对 JavaScript 进行混淆和压缩，使之没有可读性，但是这种混淆和压缩很容易被逆向工程。CSS 的压缩

1 其实这句话并不完全正确。如果您使用 SASS 编译成 CSS，或者 CoffeeScript 编译成 JavaScript，或者用 Jade 编译成 HTML 的话，用户就看不见源代码了。

空间更小，因为选择器是与 HTML 相关的，无法改变，只能去掉一些注释和空行。更多的时候，压缩代码是出于性能的考虑。为了优化页面性能，图片需要合并成 CSS 拼图方案，而如果后面的版本中图标有新增或者修改，合并图也需要对应地修改。

这一系列工作，统称为构建（build）。

构建需要环境和工具。App 工程师的开发环境比较单纯，开发者有配套的编辑器 Xcode 和集成开发环境。iOS 工程师只需要在 Xcode 里新建一个 iOS App 工程，然后引入需要库，编写好代码，就可以运行打包上传了，不需要特别考虑架构或者流程。

在前端开发中，没办法像 App 开发那样，把整个站点打个压缩包，直接丢给浏览器，说："这是整个站点，您去看吧。"

所以假设现在我们已经有了好的版本管理，能够让很多工程师有条不紊地持续提交代码到主干。并且我们有了好的依赖管理，可以把第三方的库都按照特定版本号下载到本地。最后我们需要的是构建工具，让我们能把源码最终构建成可发布的站点。

首先需要良好架构

对外的前端优化原则中，很多要求是与对内的代码管理相违背的，比如"合并 CSS，不使用 import，减少请求和阻塞"，我们不能让团队的工程师在写样式的时候，都去修改一个相同的文件。

就像是硬币的两面，一面朝浏览器，关注性能、缓存、减少重复、保持一致；一面朝前端团队，关注维护、发布流程。

对于小型站点，也许确实就是这样做的，在源代码中就已经把 CSS 和 JavaScript 合并到一起。但是站点超过一定规模，维护就麻烦了。

举一个简单的例子，如果您有一个书架，那么您的几十几百本书怎么放进

去其实没关系，您都能找到它。如果您有一个图书馆，那么您一定要有很好的规划，比如小说放在一起，工具书放在一起，然后就能方便地进行查找。如果希望管理库存，让一个读者进入图书馆的时候知道这本书是否还在，那就要进行编号，进行数字化的管理。此外，图书的种类和数量会动态变化，小说过多了超出一间房怎么处理？……为了处理这些问题，所以诞生了图书馆学。

如果只是一个小站，那么您的样式、图片怎么放其实没关系，就算丢在一个平级的目录中，您也能找到它。但当您有一个十人合作的团队，一起来管理那么多的静态资源，那么您作为架构者一定要有很好的规划，或者作为实践者也要知道如何找到团队规范。拿 QQ 空间为例，QQ 空间的个人中心每天有 6 亿 PV，个人中心的版本发布频繁，忙的时候每天更新三次版本，闲的时候两天更新一次版本。十个前端开发者中的每个人都有权限并且可能发布个人中心代码，那么如何协作呢？

所以说，“架构”是当项目变复杂之后必须考虑的问题，而项目总是会变得复杂的。小型团队一样需要合理的架构，这样可以增强扩展性。当小团队渐渐发展为大团队，网站渐渐拥有千万用户的时候，整个流程更加需要自动化。如果说求职的时候“精通 CSS”是考察开发者的手工技艺，那么持续集成就是让开发者团队可以无缝工作。

> 在我们多年来为财富 500 强公司服务的经验中，从来没有看过商业计划是完工的或者是最新的，我们也从来没见过在 12 个月的周期内没有做过大量修改的内容。——《Web 信息架构》

以我的经验，好的架构有以下几个要素。

- **有合适的分离粒度**

在旧版本的 QQ 空间中，我们的组件分离粒度太粗犷，一些小的组件没有提取出来，所以在全局上看很不一致。在新版本中我们提取出了几乎每一种组件（包括按钮、等级图标等），现在能保证很高的一致性。但是在全局

组件发生修改时，下一次构建要花半小时的时间。

- **最小知识原则**

一个组件或者对象不应该知道其他组件或者对象的内部实现细节。在 QQ 空间中，我们的配色组件跟其他组件是完全分开的，二者没有依赖关系。

- **DRY**[1]

特殊的功能只能在一个组件中实现，在其他的组件中不应该有副本。这是我们的一个严格要求。

- **最小化预先设计，只设计必需的内容**

因为好友选择器可能出现在全站任何地方，所以我们在设计好友选择器组件的时候，就留下一些自定义的空间，让组件的宽度和高度都可以直接修改，而内部元素也能自由排列。

- **通过良好的层级，让文件易于找到**

有时候我们会基于项目来区分代码库，有时候为了让多个项目共享组件和代码，会使用扁平的目录层级。我们需要根据项目具体的情况来选择。

- **在代码层面，有一致且可执行的命名规则**

从路径名到文件名都有一致的前缀、后缀、版本规则。整个团队有一致的命名风格和注释风格。

Make

Make 是一个相对古老而又强大的工具，它的第一个版本于 1976 年诞生在贝尔实验室，之后由于集成在各个版本的 Unix 系统中，所以得到了广泛应用。在将 Make 引入 Unix 开发之前，软件的编译需要运行很多 make

1　不要重复您自己（Don’t Repeat Yourself）。

和 install 脚本命令。Make 的目的是把大量的脚本命令组合起来放在一个 makefile 文件中，工程师只需要运行这个 makefile 就可以把多个目标编译到一个文件中，彻底从依赖跟踪的繁琐工作中解脱出来。

Make 经历了很多版本的重写和衍生，大部分都保留了 Make 的标准接口，然后加上一些自己的特性。常用的版本有 BSD Make 和 GNU Make 等。

Make 使用 makefile 文件来列出需要执行的任务、每一个任务依赖的资源，以及怎样完成每一个任务。下面是一个简单的 makefile 示例。

```
a.txt: b.txt c.txt
    cat b.txt c.txt &gt; a.txt
```

第一行的 `a.txt` 就是“目标”，也可以认为是我们需要生成的目标文件。冒号后面由空格分开的两个文件，它们 `b.txt` 和 `c.txt` 是“前置条件”。说白了就是依赖关系，不过“目标”不一定是文件名，可能是一个任务名。相应的，“前置条件”也不一定是文件名，也有可能是子任务。

第二行“命令”，它以一个制表符（tab）开始，然后是一句 Shell 命令，意思是把 `b.txt` 跟 `c.txt` 中的内容合并输出到 `a.txt` 中。

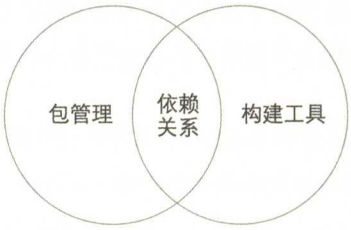
依赖关系

Make 是一个经典的构建工具，现代很多构建工具（比如 Grunt、Gulp 等）都参考了它的一些基本原则来设计。Make 的基本模型是：定义一个任务时，首先声明依赖关系，然后说明根据这些依赖，调用哪些应用程序，来生成目标文件。因为每一步都需要使用不同的应用程序，调用不同的数据，所以这里面需要设置依赖关系。

另一方面，使用包管理工具可以把项目需要用的第三方包，以及每一个包的特定版本，都集中在一个配置文件中。此后，我们通过一句命令，就可以下载这些包到本地的开发环境。每个软件包都会涉及其他的软件包，软件包里程序的运行需要有一个可执行的环境（要求有其他的程序、库等），

软件包依赖关系正是用来描述这种关系的。

所以，“依赖关系”既属于“包管理”，同时又属于“构建工具”。



包管理和构建工具都需要依赖关系。

Grunt 和 Gulp

Make 很强大，而且在全世界范围内几乎所有的计算机领域用了几十年，它的稳定可靠经过了广泛验证。不过从学习成本角度来说，它需要学习者具备一些 Linux 编程的基础，难度较高。所以，Grunt 和 Gulp 诞生了，它们都是用 JavaScript 来实现的构建工具[1]。

Grunt 有各种插件，图片来自 gruntjs.com。

Twitter 的著名框架 Bootstrap 的最初版本，曾经使用 Make 来构建项目站点，并输出编译码，现在（Bootstrap 2）已经转为用 Grunt 来构建了。

1 “任何能够用 JavaScript 实现的应用系统，最终都必将用 JavaScript 实现。”这就是著名的 Atwood 定律。

Grunt 的标题是“JavaScript 世界的构建工具”，有一个活跃的社区为它提供各种功能插件，可以执行压缩、编译、单元测试和清理等工作。

Grunt 引爆了前端架构工具的概念，得到了广泛的应用。现在，Grunt 的生态环境已经非常庞大，越来越多的开发者着手 Grunt 开发，为它添砖加瓦。但是 Grunt 有几个问题。

- 配置项过多。每一个插件的使用都需要配置输入项和输出项，使用比较繁琐。
- 子任务间的协作基于文件。基于文件的坏处是，后一个子任务必须等前一个子任务的过程完全结束，才能开始它的流程，这样比较慢。而且磁盘读写速度远远慢于内存读写。

所以虽然 Grunt 有先发优势，但是由于它有几个痛点没有很好地解决，所以又诞生了 Gulp。

在英文里面，Gulp 的意思是“大口吸”，它最初的 logo 是一杯饮料，上面有一根吸管，很形象地跟它的宣传语相呼应：“基于流的构建工具[1]”。**与 Grunt 最大的不同就在于，Gulp 基于“流”的理念。**

Gulp 基于 Node.js 的流的概念，所以前一个任务的输出就是后一个任务的输入。

从语法风格上来讲，编写任务的过程更像是“编程”，而不是“编写配置”。Gulp 通过对接前一个任务的输入和后一个任务，就像一个管道，二者可以同时进行，不输出在磁盘中，没有多余的中间产物，性能更加高效。

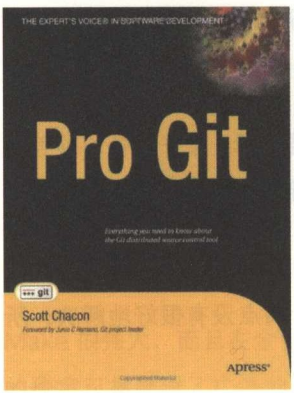
当前，Gulp 的社区还远不如 Grunt 成熟，有些功能的插件，Gulp 可能就没有。不过从个人偏好来看，我更倾向于 Gulp，它的技术理念更好。

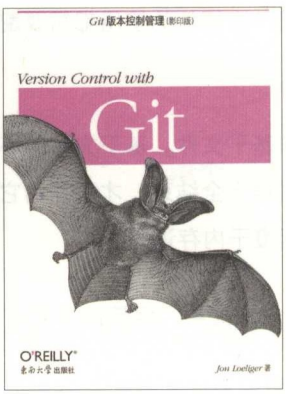
1 The streaming build system.

延伸阅读

1. 《Pro Git book》(美) Scott Chacon / Ben Straub，Apress

2. 《Git 版本控制管理》(美) Jon Loeliger/ Matthew McCullough，人民邮电出版社





理解编程语言

对编程语言的学习和讨论有两种截然相反的论调。一种说法是，所有的编程语言都一样，不存在高下之分，真正重要的是算法和对系统的理解。这种说法可能来自某个沧桑的老工程师，他号称自己用过所有的编程语言，最终达到“只要心中有剑，手中无剑胜有剑”的境界。另一种说法是，编程语言很重要，PHP/Java 是最好的语言，证据就是，大多网站使用 PHP 语言，如果它不是如此优秀，为什么这么多人会使用呢?

《黑客与画家》的作者 Paul 会告诉您，真相处于两种极端之间。编程语言有差别，但没有那么极端，也不存在绝对标准。

编程语言是什么

全栈工程师最佳实践

脚本语言的优势

编程语言是什么

机器能理解的指令叫做“机器指令”（Machine Instructions），它是 CPU 能直接识别并执行的指令，其表现形式是二进制编码。

由于机器指令与 CPU 紧密相关，所以，不同种类的 CPU 所对应的机器指令也就不同，而且它们的指令系统往往相差很大。但对同一系列的 CPU 来说，为了满足各型号之间具有良好的兼容性，新一代 CPU 的指令系统必须包括先前同系列 CPU 的指令系统。只有这样，先前开发出来的各类程序在新一代 CPU 上才能正常运行。

在最早期的时候，工程师把代码转化成二进制，再打成纸上的小孔，让计算机去读取。用机器语言编写程序是早期经过严格训练的专业技术人员的工作，普通的工程师一般难以胜任。现代工程师不再直接编写机器指令，因为二进制本不是设计为给人读写的，既不好修改也不容易查错，由于指令少，代码量也十分大，最后还要考虑不同 CPU 指令集的区别。

于是编程语言诞生了，它给每一步操作一个“别名”，也可以定义算法和数据结构，同时封装了一些复杂的操作。编程语言的目标对象是工程师，所以在功能上和可读性上不断提升，以增加语言的竞争力。工程师编写出源代码之后，通过编译器编译成目标码，就可以让机器执行了。

将编程语言编译成机器码有两种传统的方法：使用编译器（compiler）预先编译，或者使用解释器（interpreter，也叫直译器）一边编译一边运行。

编译器的工作方式是，通过编译程序直接将我们的源代码翻译成当前系统环境下 CPU 可执行的机器码，下次只需要执行这些翻译完成后的机器代码即可。这种方式的好处是执行效率非常高，因为我们最终生成的程序与直接用 0、1 写的程序并没有太大的区别。但是这种方式的缺点就是可移植性很差，很显然您不能把在 Windows 系统下编译连接成的 EXE 文件拿到 Linux 系统中去执行，Xcode 编译的 App 也只能在 iOS 和 OSX 系统中

运行。

解释器的工作方式是：将工程师编写的代码，一句一句解释给 CPU 执行。解释器不会一次把整个程序翻译出来，而只像一位“中间人”，每次运行程序时都要先转成另一种语言再作运行，因此解释器的程序运行速度比较缓慢。它每翻译一行程序叙述就立刻运行，然后再翻译下一行，再运行，如此不停地进行下去。这种方式的一种好处就是平台无关性，我们只要写出程序代码，那么它在任何操作系统、任何环境中都可以执行，当然代价就是极低的效率。第二个优点是，这个“编辑 - 解释 - 除错”的循环通常比“编辑 - 编译 - 运行 - 除错”的循环省时许多。

编译器跟解释器可以以某种形式结合，就是“运行时编译”（Just-In-Time compilation，JIT）。JIT 既有编译器的高性能，也保留了解释器的灵活性。JIT 引入了动态编译、动态再编译和递增编译等特性，它在初次编译时可能需要的时间比较久，不过以后的编译效率往往比普通编译器更高。

故事接龙

我首先想声明的是，软件工程师的工作成果（软件或者系统）完完全全是精神的产物。

原始人拿到一个锋利的石头片，然后制作了一把石刀，在这个过程中，锋利的石头是大自然已经存在的，没有人能凭空想象出一块锋利的石头片。而制作石刀的过程，是原始人看到已经存在的材料，或者工具，然后想象出石刀这个武器，于是制作了出来。这个产品的一半是自然的产物，一半是精神的产物。

随着人类智慧的进步，我们开始制作出完全精神的产物，比如《禅与摩托车维修艺术》里提及：

> （人们）把金属制成各种形状——管子、杆子、工具、组件——把这一切都组合起来，但不能违背它运作的理论，然后让它们以实体来运作。然而从事机械铸造、打铁或是焊接的人则不认为钢有任何形状，如果您有很好的技巧，钢就能变化出任何形状，如果您技巧不够的话，就做不出来了。如果您想做成梃杆，就必须有这种技巧，而它的形状是您设计的。这一点很重要。钢铁？钢铁也是人所设计出来的，因为在自然界之中并没有钢铁的存在……一切都存在于人的心中。

虽然软件是完全精神化的产物，但这并不是说整个系统都是以我们一己之力构建的。我们会基于他人的精神产物来工作，比如其他人设计的工具和组件。我们可能会以为它们是自然存在的，我们只是“使用”和“遵循”它们，但是您应该知道，这些都是他人的精神产物。您的软件作品，积累了所有的计算机架构师、系统设计者、语言设计者、编译器设计者……上万人的想法在里面。就好像我在写书，也凝聚了我阅读过的所有作者的精神积累，他们都对我有影响。

所以，作为全栈工程师，理解您所工作的平台和编程语言背后的特质，是非常有用的。不同的编程语言差别很大，它们有不同的抽象偏好，有不同的设计思想，有不同的语法风格，有不同的依赖环境，背后有不同的人在营销。

您跟编程语言的设计者们好像在玩一个故事接龙的游戏，编程是您们之间的一个互助的过程。您真正理解了设计者讲的故事，才能把您自己的这一段故事讲好。

语言的进化

编程语言是脱离于操作系统和软件平台的，它只是对软件功能的实现规格说明书。但是很多语言会跟某个平台联系十分紧密，这一般是营销上的考虑。比如 JavaScript 是浏览器中支持度最高的编程语言，已经在前端编程中成为垄断和标准。所以它也在扩张到其他领域，已经有人开始大力推广 JavaScript 在服务器端（Node.js 和 IO.js）和桌面端的应用（Node-Webkit）方案。

语言的普及率本身也会造成竞争优势。编程语言用户群扩展存在马太效应，越是普遍和用户群广泛的编程语言，越会带来更多的新用户。

一个原因是，大量的开发者用户会发明很多方便的库。库很吸引人，因为您可以直接用它来完成某个复杂的功能。大量的开发者活跃在 StackOverflow 社区，这种语言的问题会有很多人帮您解答。

另一个原因是，大量开发者使用的语言，更容易被公司作为技术选型。因为开发者多，所以在招聘时会有更多的候选人。

PHP 非常简单，易学易用，好读好调试，因此新工程师成长很快。PHP 是一种脚本语言，其好处是编程效率高，能够支持产品的快速迭代。很多性能较差的免费主机和便宜的虚拟主机都支持 PHP，所以有很多 PHP 开发的开源博客系统和 CMS 框架。这可能是 PHP 在语言设计之初就决定了的，因为它最初的名字就是个人主页（Personal Home Page）的缩写，不过后来被改为“PHP：超文本预处理器”（PHP：Hypertext Preprocessor）。

但与传统的编译语言相比，脚本语言的 CPU 和内存使用效率不高，而且它没有命名空间，不一致的函数命名规则等让一部分程序员认为它不够优雅。

在 Facebook 的创立之初，大量业务都使用 PHP 编写，这让他们可以快速地开发。不过随着用户量越来越大，性能渐渐跟不上要求。要优化性能，常见的办法是直接用 C++ 重写 PHP 应用中比较复杂的部分，作为 PHP 的补充。实际上，PHP 已经转变为一种胶水语言，连接前端 HTML 和 C++ 应用逻辑。

另外，Facebook 还集成了 HPHPc、HPHPi、HPHPd 以及 HHVM 这 4 种脚本引擎，开发出 HipHop for PHP。HipHop for PHP 可以将 PHP 转化成 C++ 程序，从而大大减少了服务器的压力，能够让同一台服务器多承受 100% 的用户增加。为了让这一系统也惠及社区，他们将 HipHop 开源，希望能够进一步提高更多大型复杂 PHP 网站的可伸缩性。

所以，语言的特性会吸引用户群，用户群也会反过来影响和改变语言本身。

首选语言之争

随着计算机性能越来越强，价格越来越便宜，一个越来越明显的趋势是，软件开发流程中有了越来越多的软件层。

在 Windows 发布之初，它的原生应用程序是用 C 语言和原始的 API 编写的。之后，开发者的选择越来越多，比如可以使用 Microsoft Visual Basic 来提高开发效率，以及支持图形化界面开发。Visual Basic 是微软开发的编程语言和集成开发环境（Integrated Development Environment，IDE），用它能够开发基于 Windows 平台的原生应用程序，可以说是首选语言。不同操作系统的首选语言都不一样，拥有不同的底层 API。Windows 应用程序就像其他原生程序一样，具有排他性，它不可能在 Linux 或者 OS X 中运行。

那么如果想要创建在 Windows、OS X 和 Linux 下都能运行的应用程序，怎么办呢?

Java 可以实现这个目标。

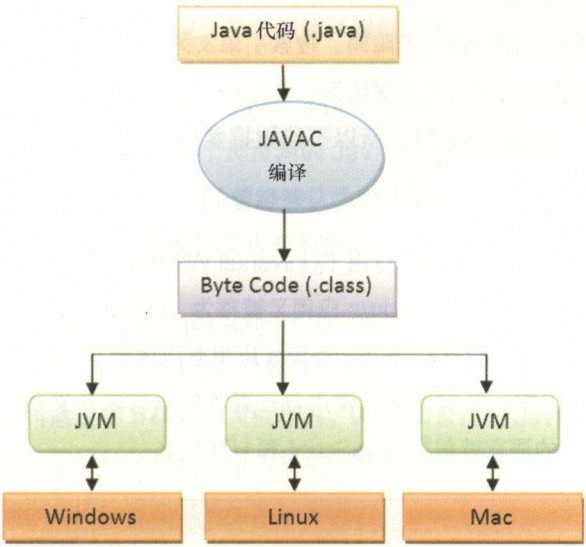
“一次编写，到处运行”（Write once, runanywhere、WORA），是 SUN 公司用来展示 Java 程序语言的跨平台特性的口号。理想中，这意味着 Java 可以在任何设备上开发，编译成一段标准的字节码并且可以在任何安装有 Java 虚拟机（JVM）的设备上运行[1]。

由于编译码（.class）是在 JVM 上运行，而不是在操作系统上直接运行，所以可以认为 Java 是 Windows、OS X[2] 和 Linux 等操作系统上的次优语言，

1 美中不足的是，JVM 在不同的操作系统上有多种不同的实现，导致 Java 程序在不同的 JVM 虚拟机和操作系统上执行的时候有微妙的差别，所以一种应用可能需要在许多平台上进行测试，这造就了一个 Java 开发者的笑话：“一次编译，到处 Debug”（Write Once, Debug Everywhere）。

2 OS X（前称 Mac OS X）是苹果公司为 Mac 电脑开发的专属操作系统。

而不是首选语言——原生应用程序。这种情况下，它的性能直接由 JVM 决定，通常会比原生程序差一些。



Java 可以在任何支持 JVM 的设备上运行。

类似地，网络中的 JavaScript 由浏览器来运行，而不是用户操作系统直接运行，它也是操作系统中的次优语言。网页中 JavaScript 的性能除了受开发者编写的影响，还受浏览器的执行效率影响。Chrome 之所以广受好评，有一个原因就是它使用高性能的 v8 引擎，执行 JavaScript 效率非常高。

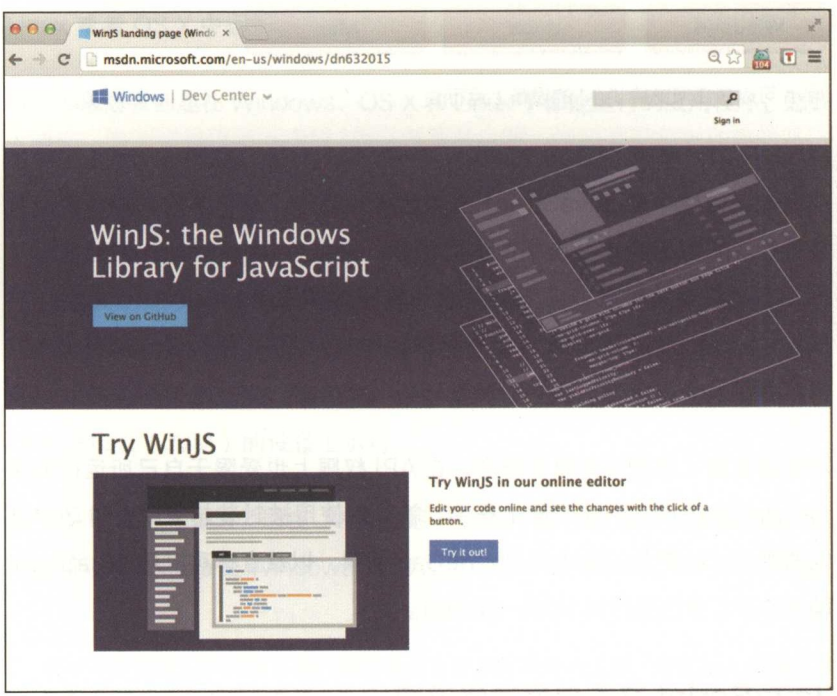
次优语言除了效率不如原生程序，在 API 权限上也受限于自己所运行的平台，所以如果用它们编写原生程序，就只能使用被封装起来的 API 了。其他的例子可以参考 iOS 中的 PhoneGap 框架，以及宣称可以用 JavaScript 编写跨平台桌面程序的 Node-Webkit。

JavaScript 并不总是次优语言

在 Web 前端中，JavaScript 是首选语言。我认为 Web 前端是一个影响力

非常大的系统，它的开放、流式特性、跨设备跨平台兼容性等特点是它的优势。甚至在不同的浏览器中有不同的表现，这也是 Web 前端的一个重要特性。如果把 Web 前端视作一个系统，HTML、CSS、JavaScript 是首选语言，它们具有诸多优点：高性能、无障碍、搜索引擎友好等。而 Flash 等需要插件支持的编程语言，就变成了次优语言，它运行在 Flash 插件中，所以性能比较糟糕。它不具有原生特性，所以不能被搜索引擎理解，对视障用户不友好。

在 Windows 8 以上的“通用 Windows 应用”(Universal Windows App) 中，JavaScript 也是首选语言。通用 Windows 应用又被称为“Windows 运行时应用”(Windows Runtime App)，在 Windows 应用商店或者 Windows 手机应用商店里出售和下载。开发者可以使用自己喜欢的语言来开发通用 Windows 应用，可使用的语言就包括 JavaScript 和 HTML5。



WinJS：使用 JavaScript 编写通用 Windows 程序，截图来自 msdn.microsoft.com。

语言的性能

语言只负责描述一个程序，而程序运行的速度，其实很大程度上不取决于语言，而主要取决于算法和编译器的质量。

以 JavaScript 为例，JavaScript 语言以 ECMA-262 规范作为语言定义，而具体的解释器由各个浏览器的 JavaScript 解释器（也叫 JavaScript 引擎）来解释。JavaScript 曾经被认为性能不佳，但是由 Chrome 公司开源的 v8 JavaScript 引擎带来了很高的性能。

v8 引擎与传统 JavaScript 引擎的解析流程有所不同。传统 JavaScript 引擎通常先把 JavaScript 代码编译成字节码（Bytecode，是一种中间码），然后再通过字节码转译为机器码。v8 引擎直接把 JavaScript 代码编译成机器码，所以性能得到了很大提高。而且 v8 还使用了一些其他的优化技术，比如优化的垃圾回收器和缓存策略。

编译器和语言基本上是两码事。同一个语言可以有很多不同的编译器实现，每个编译器生成的代码质量都可能不同，所以您没法说“A 语言比 B 语言快”，只能说“A 语言的 X 编译器生成的代码，比 B 语言的 Y 编译器生成的代码高效”。这几乎等于什么也没说，因为 B 语言可能会有别的编译器，使得它的代码更高效。

语言的设计理念

在管理界有两种假说。

一种假设“人性本恶”，即所有的员工都讨厌工作，只要没有监督，就一定会偷懒怠工，绝大多数人没有雄心壮志，怕担责任，宁可被领导骂。多数人必须用强制办法乃至惩罚、威胁，使他们为达到组织目标而努力。这叫做 X 理论。

另一种假设“人性本善”，即所有的员工都热爱工作，只要给他们创造

舒适的环境、合理的授权和适当的激励，他们就会发挥很高的生产力，能力的限制和惩罚不是让人为组织目标而努力的唯一办法。这叫做Y理论。

这很有意思，因为持有这两种世界观的员工和管理者都存在。所以相信X理论的管理者会设置很多的规章制度——要求上下班打卡，限定工作范围，重视金钱奖励，同样也重视惩罚。支持Y理论的管理者会提供生理需求之外的福利——重视鼓励和授权，扩大工作范围，设置灵活的上下班时间，等等。

回到编程语言的话题。有些语言设计者假设使用者能力不够，会尽可能地犯错，把系统架构搞乱，所以他在语言中增加了一些限制。有些语言的设计者假设使用者非常聪明，知道自己在做什么，所以往往在语言中提供更多的灵活性。

管理界经过科学的实验和调查发现，与传统工厂或者机械劳动厂商中X理论发挥更大效用相比，在互联网公司或者智力活动占多数的公司中，Y理论更加适用。从表面上看，Y理论和X理论是相互对立的，但实际上它们是同一个问题的两个侧面，而不是互不兼容的必选其一的对立关系，一味强调一个方面显然是片面的。X理论和Y理论是统一价值杠杆上的两个不同终端，优秀的管理者会在杠杆上选择一个最适合自己团队的管理方法。

对于编程语言的灵活性同样如此。一些灵活的特性，比如动态类型，它的优点是更加灵活，可以在运行时改变变量的类型，缺点就是调试错误时更加困难。静态类型比较安全，不会出现类型错误，但缺点是需要写更多类型相关的代码，不便于阅读，不清晰明了。

思考一下，您使用的语言是哪一种设计理念呢?

全栈工程师最佳实践

通用用途语言 vs 特定领域语言

很多编程语言倾向于通用解决方案，而不是只解决具体问题。这些语言都被设计为可以在任何领域使用，比如 C、Java、Python 和 XML，它们被称为“通用语言”（General Purpose Language，GPL）。我们可以看到用 C 编写的所有类型的软件，从游戏到客户端软件，从服务器端软件到手机端软件。

与之相对应的，有些编程语言被设计为特定领域专用，叫做“特定领域语言”（Domain Specific Language，DSL）。DSL 的目的是解决特定领域的问题，而不是像 GPL 一样可以解决任意的软件问题。DSL 在计算机软件开发中十分常见，比如前端开发中常见的 HTML 和 CSS 就是一种 DSL，专用于 Web 开发。MySQL 是一种 DSL，专用于操作数据库。Make 是一种 DSL，专门用来处理 Shell 脚本操作系统文件输入和输出。

有些人认为 CSS 不是一种 DSL，因为它不是编程语言。但是 DSL 并没有说一定是编程语言，标记语言、模型语言和编程语言都属于它的范畴。

有些人认为 DSL 是解决特定领域问题的好办法，他们鼓励任何人创建自己的 DSL。但事实上，我很难想象出必须创建自己的 DSL 的很好的理由。这个世界上已经有太多 DSL 了。对于想深入研究编程语言的专家，当然可以创建自己的 DSL，还可以创建自己喜欢的脚本程序语言和解释器，都没有问题。但是如果您是一个以解决问题为目标的全栈工程师，我建议您在考虑发明一个 DSL 之前先考虑以下方案。

- 尽量用您熟悉的通用语言来解决问题，比如 Python、Java 或 C++。
- 优化您的解决方案，提炼出一种真正精简、优雅的扩展库。
- 开源您的扩展库，根据其他人的贡献来继续优化解决方案。

- 如果想简化配置文件的语法，可以创建一个脚本包装器来专门为库工作，这就是您自己的 DSL[1]。
- 如果最后您还是想进一步优化下去，那就发明您的 DSL 吧。

框架和库拓展了语言

Ruby 是世界主流编程语言中唯一由亚洲人开发的编程语言。日本人松本行弘（Matz）于 1995 年正式发布 Ruby，所以早期的非日文数据和程序都比较贫乏，在网上仍然可以找到早期对 Ruby 的数据太少之类的批评。约于 2000 年，Ruby 开始进入美国，英文的资料开始发展。

2004 年，Ruby 的框架 RoR（Ruby on Rails，Rails）发布。Rails 一经发布就迅速在开发者之间传播开来，Ruby 更加广为人知。Ruby 更于 2006 年获选为年度编程语言。

Matz 公开承认，Ruby 之所以有现在的人气基本上都是由于 Ruby on Rails 的贡献。而 Rails 成功的原因，首先得益于 Web 的快速发展，几乎所有的软件开发平台都在瞄准 Web 这个领域。以往使用 CS（Client-Server，客户端 - 服务器）架构开发的系统，现在都可以在 Web 上实现了。在 Web 上能够开发的应用变多了，这是一个主要的背景。其次 Ruby 语言的一些灵活的特性，比如元编程、高可扩展性，让开发 Rails 成为可能。DHH[2] 曾经打算用 PHP 来开发一个这样的框架，但是 PHP 在元编程方面有很多限制，所以 DHH 最终转向 Ruby 来开发。

在快速开发中，真正重要的是库，所以在 Rails 出现后，因为“我要做个网站，Rails 最快”这样的理由而使用 Ruby 的人变多了。好的框架让语言得到了更广泛的关注和使用。

全栈工程师的目标往往是快速解决商业问题，不一定需要长期完美的方案。

---

1 又被称为内部 DSL，好处是可以使用您熟悉的语法，无需学习一种新的语言。对于动态语言（比如 Lisp、Ruby 和 Smalltalk）来说，这一点比较容易做到。

2 大卫・海纳梅尔・韩森，丹麦软件工程师，以创建了 Rails 而闻名。

使用方便好用的框架能大大节省学习成本和开发时间，所以有些时候我们的技术选型步骤是：先选择框架，然后选择语言。

Twitter 在初期就使用 Rails 快速开发，那时候没人知道 Twitter 会获得今天的成功，人们觉得，这种只能写 140 个字的博客有什么意思呢？但 Twitter 却出人意料地获得了巨大成功。在这个过程中，Twitter 增加了很多新功能，在它快速发展的过程中，Ruby 的贡献是相当大的。因为一个新功能从构思到付诸实现，可以用很短的时间就能够完成。

这就是正确的技术选型带来的好处。

但是后来 Twitter 的访问量越来越大，最初的架构已经无法满足需求，所以后面他们改用 Scala 来重写 Twitter。一方面原因是 Scala 是编译语言，性能更好，另一方面也正好趁此机会编写全新的架构。

脚本语言的优势

最近，苹果公司发布了一款新的开发语言 Swift，有些人看到它的语法很像 JavaScript，高兴得直呼："脚本语言全面逆袭的时刻到了！"

其实这是一个误解，Swift 是一种语法很像脚本语言的编译语言。脚本语言跟编译语言的差异不在于语法，而在于编译机制。

脚本语言（script language）是指支持用脚本的方式编写程序的语言，它无需编译即可直接在运行时环境中解析。在操作上，它缩短了传统的"编写 - 编译 - 链接 - 运行"过程。脚本语言通常具有简单、易用的特性，而且常常很短小。

相比编译语言脚本语言有更高的开发效率，但是在执行效率上会有所牺牲。由于现在的趋势是硬件成本越来越低，而工程师的人工成本越来越高，所以脚本语言的使用空间越来越大，有一些脚本语言（Python、Ruby）已经在成熟的商业网站中使用。

不同的脚本语言有不同的设计原则，但是它们往往有一个共同的目标，就是以简单的方式，快速完成某些复杂的任务。

脚本语言不需要编译

脚本语言的特点是无需编译即可运行，它在对应的运行环境中直接运行，运行时通过解释器来逐句解析。

因为语言跟对应的解释器（或者编译语言跟对应的编译器）是分开的两个概念，所以从科学上讲，只要给定合适的运行时环境和库支持，任何语言都可以作为脚本语言来使用（也就是编写脚本）。也就是说，“编写脚本（scripting）”是对语言的一种使用方法，而称某种语言为脚本语言是一种工程上的约定俗成的用法，而不是科学上的定义。

而且另一个问题是，无论是脚本语言还是编译语言，最终都需要编译成机器码让机器来执行。比如 JavaScript 语言，在 v8 引擎中被编译为机器码然后执行，如果是使用 Node.js。那么这个机器码可能会被缓存起来[1]，这样的话，跟编译语言就没什么区别了。

有点绕是吗？那么以举例来定义脚本语言吧。通常来说，我们提到脚本语言的时候，是指动态的、高抽象级别的通用语言，比如 Perl、Python 和 Ruby 等，它们常常用来编写简单的程序（最多几千行代码），但也不排除构建大型软件的可能性。脚本语言也用来指某些特定领域的语言。

是否需要编译可以说是脚本语言的唯一判断依据，那么，还有哪些方法可以帮助我们理解脚本语言呢？或者说，为什么我推荐学习脚本语言呢？

脚本语言常常不用关心清理内存

因为脚本语言的设计目标是快速写出能运行的程序，它更倾向于取悦工程

1 即前一节中介绍的 JIT 机制。

师，而不是优化性能。所以在语法上就忽视内存管理，而该语言的解释器则各显神通，把清理内存垃圾的重担揽在自己的黑盒里面，无需工程师关注。

比如 IE、Chrome、Safari 各有自己的 JavaScript 引擎，最新发布的 OS X Yosemite 系统中的 Safari 声称自己的 JavaScript 引擎速度超越 Chrome v8，可能就是在垃圾清理方面有更好的优化。

这条规则不一定只对脚本语言有效，所以我用到了“常常”。编译语言有时候也可以不用关心垃圾清理，比如 Objective-C 是基于 C 的扩展语言，它是一个典型的编译语言，它的官方运行环境是 Xcode。Xcode 5 可以通过在项目选项中勾选“ARC（Automatic Reference Counting）”来启用自动引用计数功能，以此达到让编译器控制垃圾回收的目的，这样工程师就不用手动释放内存，因为编辑器能探测到对于哪些对象已经没有任何引用了，就可以自动回收内存。

脚本语言常常会针对特定领域优化

以 PHP 为例，PHP 是一门服务器端的脚本语言，它最开始被设计为针对 Web 开发，有时候也被视为一门通用语言。

由于 PHP 针对 Web 开发进行优化，并且它本身的语法也是非常简单，所以这门语言在 Web 开发领域获得了巨大的市场占有率（根据 2013 的抽样统计，世界上有 39% 的网站使用 PHP 语言作为服务器端语言）。那么它做了哪些优化呢?

举例来说，在 PHP4.1 中，引入了全局变量 $_GET、$_POST 和 $_SESSION，可以直接获取请求数据中的 get、post 或者 session 数据。作为一门语言规范（而不是第三方库）直接支持这样的数据，可以说是把 Web 开发的成本大大降低。所以现在很多语言社区都会有这样开玩笑的帖子：“PHP 是世界上最好的语言。”

脚本语言常常是动态类型语言

静态类型语言在声明变量时需要声明变量的类型，并且不可改变，比如以下 Java 代码：

```
String text = "programmer";
text = {'p', 'r', 'o'}; // 编译错误
```

动态类型语言不用声明类型，比如以下 Ruby 代码：

```
text = "Hello"
```

静态类型语言在编译时就知道每一个变量的类型，如果类型错误，那么就会报语法错误。动态类型语言编译时不知道每一个变量的类型，如果类型错误，比如对数字 a 调用 a[1] 就会报运行时错误。

值得一提的是，某些静态类型语言具有类型推论（type inference）特性，可以根据程序上下文判断变量类型。比如 Scala 是静态语言，但是可以通过以下语法创建 String 变量：

```
val text = "Hello"
```

虽然没有具体写出 String，但可以通过“Hello”来推断出 text 是 String 类型。

脚本语言常常是动态类型语言，好处是语法简洁，最直接的效益就是节省打字的功夫。代码具有较高的弹性，Ruby 中常用的“鸭子类型[1]”就是动态类型的一种很好的应用。

而且 Ruby 允许您对官方的类和模块进行无条件的修改和扩展，这种灵活性导致了一些很有趣的情况，比如在 Rails 中您甚至可以使用以下代码表示 20 小时之前：

```
20.hours.ago
```

1 在程序设计中，鸭子类型（duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

这种表达非常适合英语思维的人，而且似乎短得有点不可思议。但是了解背后的原理之后，就觉得恍然大悟了：首先扩充整数类中的 hours 方法，对 20.hours 返回 72000（3600 秒 ×20）这个值，ago 方法又返回该秒数之前这一时刻的一个 Time 对象而已。

有趣的是，Ruby 的发明人是日本人，但是在欧美有很多粉丝，他们热烈地讨论能在 Ruby 的语法框架下把 Ruby 改得多像英文。

至于动态类型语言的缺点，是在执行时才会把类型错误呈现出来；查询类型是否支持某方法也只能根据 API 来判断，运行效率较低。

脚本语言的抽象层常常更高

抽象层级更高是指更多地从计算机细节中抽象出来。比起抽象层级更低的语言，高级语言更像是自然语言、更易用，也更适用于某些特定的领域。

抽象层级的高低没有严格的界定，根据维基百科的说明，有名的高级语言包括 Java、Python、Visual Basic、Delphi、Perl、PHP、ECMA Script 和 Ruby 等，它们大部分都是脚本语言。

脚本语言常常有包管理器

从 Python 和 Ruby 在 GitHub 的开源状况来看，很多程序都是代码量很少的脚本程序，究其原因，是因为工程师可以很好地调用他人的脚本。这些小程序能很好地与其他组件通信，合力完成一个复杂任务。Linux 的编程哲学是：

> 一个程序只做一件事，并做好。程序要能协作。程序要能处理文本流，因为这是最通用的接口。

很多脚本语言都有自己特定的包管理器，用来下载脚本，或者进行依赖管

1 也叫 KISS 原则：Keep it simple, stupid。

理。Ruby 有 gem，Python 有 pip，Nodejs 有 npm，安装一个脚本是如此简单：

```
gem install mysql
```

根据上面的介绍可以看出，Swift 确实不是脚本语言，因为它会编译成二进制然后再运行。但是它再一次模糊了脚本语言和编译语言的边界。

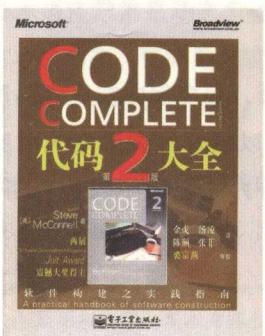
我想，Swift 是脚本语言还是编译语言其实没有那么重要，重要的是，这门语言的设计者做出了哪些选择，而舍弃了哪些选择。这样的选择，往往能体现出设计者对这门语言的理解和目标。对语言进行分析，能帮助我们进入到设计者的世界，从而写出更好的代码。

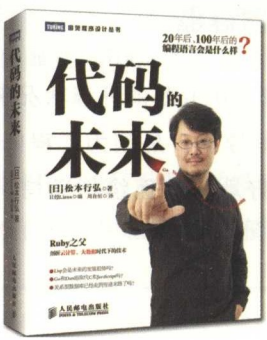
延伸阅读

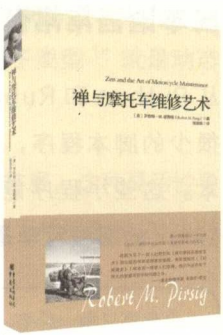
1. 《代码大全》（美）Steve McConnell，电子工业出版社

2. 《代码的未来》（日）松本行弘，人民邮电出版社

3. 《禅与摩托车维修艺术》（美）罗伯特·M. 波西格，重庆出版社







全栈游乐场

游乐场是一个孩子开心玩耍和体验新东西的地方，所以我把 VPS 比喻成游乐场，因为它是全栈工程师玩耍的地方。

VPS

实践

VPS

虚拟专用服务器（Virtual Private Server，VPS）是把一台服务器分割成多个虚拟专享服务器的优质服务。每个 VPS 都可分配独立公网 IP 地址、独立操作系统、磁盘空间、内存、CPU 资源、进程和系统配置，模拟出“独占”使用计算资源的体验。

比较廉价的选择是虚拟主机（Virtual Host），又称虚拟服务器或虚拟空间。虚拟主机将一台服务器的某项或者全部服务内容逻辑划分为多个服务单位，对外表现为多个服务器，从而充分利用服务器硬件资源。

下面举个例子，自己买来一台实体的服务器放在家里，24 小时开启，这就像是买房子。买实体服务器的好处是 100% 的控制权，缺点是麻烦，要考虑机器的更新换代，家里要准备备用电源，带宽要设置足够，如果用户量过多还要增加机器和带宽，等等。

如果我们租了一台 VPS，按年付费，这就是租房子，虽然产权不是我的，但是我拥有完整的使用权。VPS 可以像独立服务器一样，重装操作系统，安装程序，单独重启服务器。而且很多 VPS 提供升级功能，可以在最开始使用最廉价的方案，随后扩展到更高级的套餐，一键升级，非常方便。VPS 的价格介于实体服务器和虚拟主机之间。

如果使用虚拟主机，跟其他人共享 CPU 和内存等资源，这就像是合租。如果其他人在使用卫生间，您就没法用了。虚拟主机的好处是很便宜，国内一些服务提供商提供年费仅几十元的虚拟主机。

虚拟并非指不存在，而是指空间是由实体的服务器延伸而来，其硬件系统可以是基于服务器群，或者是单个服务器。

虽然 VPS 对个人开发者来说比较昂贵，但我还是推荐全栈工程师买一台 VPS，自己玩一玩。下面是我的理由。

对于网站的全貌有所了解

不识庐山真面目，只缘身在此山中。长期专注于网站中某一环节的工程师往往不理解整个网站的全貌，即使使用第三方云服务也没有改善多少。但是只要开始接触 VPS，您就需要了解软件系统的全貌。

举一个简单的例子，来看看使用第三方托管的博客服务和自己搭建一个博客网站的区别。

如果采用第三方的托管服务搭建博客系统，新建一个账号，就可以开始写了。好处是很方便，缺点是在自定义功能上（比如绑定独立域名、安装插件和修改路径格式等）没那么灵活。

如果要用 VPS 搭建一个博客网站，就麻烦一些。

- 初始化。Linode 提供一键安装操作系统，等待几分钟操作系统就安装完成了。
- 安装最新的 Apache（或者其他服务器）。启用 Apache 的 rewrite 等模块，WordPress 的 URL 重写会用到。
- 安装 MySQL 数据库，配置 WordPress 的数据连接。
- 配置域名和路由（包括访问路由配置、日志配置、网站域名和别名等）、启动服务器、查看资源利用，等等。
- 当然，也不要忘了安全防护和设置自动备份。

看上去很折腾，不过这种折腾是有意义的折腾，因为它让您在操作的过程中理解了 Web 的工作原理。

与之相比，有些折腾既浪费了自己的时间，也没有得到真正技术上的提高，比如在网上花很多时间去搜索“物美价廉”的 VPS，甚至希望能免费使用。这也是我初学 Web 开发时犯下的错，现在才明白这个世界上没有既免费又

稳定的 VPS。世界上所有的服务和商品无不如此，如果某个商品声称自己是免费又完美的，您最好离它远远的。

折腾 VPS 的学习曲线比较平坦，只需跟随教程，简单地搭建自己的第一个站点，慢慢地就熟悉基本的 LAMP 了。整个流程走下来，对网站全貌的理解会有很大提高。

以后在工作中，这个经验也许就会派上用场。前端工程师也需要配置反向代理来某个 host 能指向两台开发服务器，或者查看服务器错误日志，看看是什么原因导致开发机无法访问。

您观察网站的视角越高，您就越能有效地定位问题和解决问题。

时间就是金钱

推荐使用 VPS 的第二个原因是稳定。我在大学的时候曾经使用过一段时间的虚拟空间，是多人共用的那种。由于一台主机上可能会挂 500 个网站——甚至更多，那么每个网站能使用到的平均性能就可想而知了（想象一下您跟 50 个陌生人一起合租一所房子）。网站宕机是家常便饭，由于邻居占用了太多的带宽，网站经常 ping 不通，这时候能约束到合租邻居的基本只有道德了。

而且虚拟空间提供商也不会提供很好的服务，如果您需要开启服务器的某个功能，比如 Apache 模块，客服给的回答基本都是“这是高级功能，您如果有这样的需求可以开个 VPS。”

使用多人共用的虚拟空间还有一个问题就是大家共享一个 IP，所以如果某个邻居由于敏感的数据导致 IP 被“墙”，所有人的网站就都访问不了了。这时候就只能等待服务商改 IP，在那之前什么也做不了。

所以，如果您真想把精力集中在有用的技术上，而不是服务器无响应或者 IP 被“墙”等一些无聊的琐事，那么 VPS 是最有性价比的选择。

部署自己的环境

如果选择虚拟空间，很多服务商都已经设定好了服务器环境，并且不能修改，比如国内某网提供的虚拟主机。

- 操作系统：Linux（Aliyun Linux 64 位）
- 支持语言：PHP4.3、HTML、WAP、Perl 5、独立 cgi-bin
- 数据库：SQLite v2.8.14（与网页空间共享大小）
- 独立网页空间：0.5GB
- 价格：580 元 / 年

对于这样的环境，网站部署是非常有局限性的，Node.js 和 Python 都不支持，甚至 PHP5 都不支持（HTML 算是什么支持语言，这是浏览器端渲染的语言，根本无需服务器支持）。

而自己部署的 VPS，从操作系统、服务器层级，再到支持的服务器语言，都有完全的控制能力。想学 Node 和 Rails？没问题，自己安装一个。

学习 Linux

购买 VPS 之后，您会经常需要通过 SSH（Secure Shell）登录到服务器，然后进行部署、修改和配置等操作。SSH 除了是一个安全协议之外，它还提供了一个 Shell 环境，您在 SSH 窗口中敲入 Linux 的命令操作，服务器就会对应地给出响应。

这是学习 Linux 使用的不错的环境，您可能觉得在 VPS 中学习 Linux 过于奢侈，其实不是的。我的观点是，它除了是一个学习 Linux 的环境，更是一个逼您去运行 Linux 的环境。

为什么要学习 Linux 呢？一个原因是，Linux 已经在服务器操作系统市场取

得了垄断地位，考虑未来云计算的发展方向，其他服务器端霸主地位不会轻易动摇。第二，Linux 的版本虽然一直迭代，但是基本思想还是相对稳定，所以收益也比较高。

理解 HTTP

我曾经有一个任务，就是在某台开发服务器上部署一个反向代理，指向另一台服务器。我当时以为是很简单地把请求转发过去就可以了，但是发现把 JavaScript 脚本直接转发过去，会有跨域的限制。请求而来的 js 文件因为域名被更改了，所以不能在当前域名下运行。所以我研究了 Apache 的 mod_rewrite 代理的各种参数，最终解决了这个问题，自己对 HTTP 的理解也更深入了。

再有一次是页面需要加载 CSS3 Web Font，但是 Firefox 总是无法正常显示。我经过一些查询就了解到这也是字体资源的跨域限制，Firefox 为了防止其他网站盗用某个网站的字体，所以限制了只有来自一个域名的自定义字体才能正常显示。要修复这一点，只要在字体的输出 HTTPS 头中设置允许所有域调用自己就可以了。如何修改呢？在 Apache 的配置文件中就可以修改。

还有很多的例子，通过自己去配置和操作服务器，会让前端工程师也对 HTTP 有更好的理解。

实践

如果您已经决定入手一个 VPS，请继续阅读。

VPS 选择

在 V2EX、知乎上，最能引起程序员热烈的讨论和争论的问题绝对是“什么编程语言最好”，排名第二的可能就是“请推荐一个 VPS”。

每个人都有不同的需求，不过选择 VPS 的原则都差不多。首先要考虑的当然是性价比，主要参数是内存、CPU、硬盘和流量。

- **内存一般是 VPS 的主要瓶颈**

内存的消耗跟用户访问量和程序优化有关。用户访问量越大，内存消耗越大；不同的程序对性能的消耗不一样。对于 PHP 程序来说，内存用量较小；对于 Rails 程序来说，内存消耗一般更大一些。建议至少选择 512MB 的内存作为基础方案，这个级别的 VPS 可以在支持一个访问量不大的 WordPress 的情况下，还有额外的内存供您做一些小实验。

- **CPU 是相对没那么重要的性能指标**

对我们大多数人来说，CPU 都不是瓶颈。Web 应用程序一般都对 CPU 不敏感，只有一些需要编译的操作会比较依赖 CPU 性能。

- **硬盘的大小和读写速度是关键**

首先，所有的资源都放在 VPS 中，需要占用硬盘空间。其次，Linux 的内存管理会在硬盘中设置 swap 空间，swap 的功能是利于磁盘空间做内存的缓存，把内存中不经常用到的数据放到硬盘的 swap 空间里去，以此释放出更多的物理内存让系统去运行更多耗费大内存的程序。但是硬盘毕竟比内存慢一个数量级，所以也不能完全替代内存。在硬盘读写中，固态硬盘比传统机械硬盘的读写速度高十几倍，而且更加稳定。

除了性能和稳定性需要考虑，**还有一个重要的考虑就是客户服务**。如果您的资金比较充裕，首先推荐 Linode，稳定的服务加上快速的客服支持，可以让您把关注点都放在技术上。新手可以选择一台最低配置的 VPS，如果您对比一下国内的虚拟主机服务，它其实具有相当不错的性价比。

曾经有一次，我发现分配给我的 IP 已经上了长城防火墙的黑名单。于是我就提交了一个工单，Linode 十分钟内就给我更换了一个新的 IP，服务质量非常好。

关注服务器安全

能力越大，责任越大。当您有了安装 VPS 操作系统的能力，您就一定要学会保护自己的服务器。Linode Library 上有一系列非常优秀的新手教程，讲述了如何通过一些简单的操作让服务器更安全。

简单来说，大体步骤有以下几点。

- 新建一个普通用户，以后都不要用 root 登录了。
- 使用 SSH 的名值对的登录方法，禁用用户名和密码的登录方法。
- 禁用 root 账户通过 SSH 登录。
- 安装一个防火墙。
- 安装 Fail2Ban，杜绝字典攻击。

操作系统选择

Linux 发行版的选择会让新手比较困惑。从性质上划分，Linux 的版本大体分为由商业公司维护的商业版本与由开源社区维护的免费发行版本。

对新手来说，CentOS 的学习上手成本是最低的。您会发现，非常多的商业公司部署在生产环境上的服务器使用的都是 CentOS 系统。CentOS 是从 RHEL 源代码编译的社区重新发布版。CentOS 设计简约，命令行下的人性化做得比较好。并且性能稳定，有着强大的英文文档与开发社区的支持。

另一种广受欢迎的操作系统是 Debian。一般来说，作为适合于服务器的操作系统，Debian 比 Ubuntu 要稳定得多，可以说稳定得无与伦比了。只要应用层面不出现逻辑缺陷，Debian 基本上是很稳固的。Debian 整个系统基础核心非常小，不仅稳定，而且占用硬盘空间小，占用内存小。128MB 的 VPS 即可以流畅运行 Debian，而运行 CentOS 则会略显吃力。但是 Debian 的帮助文档相对于 CentOS 略少，技术资料也少一些。

Ubuntu 的操作跟 Debian 类似，Fedora 跟 CentOS 类似。如果您不知道如何选择，那就选择您的 VPS 提供商支持的 CentOS 或者 Debian 的最新版。

域名解析

一般来说，域名购买商（比如 GoDaddy）和服务器提供商（比如 Linode）都提供 DNS（Domain Name System）解析的能力，不过，域名在哪里注册和域名在哪里解析是两回事。

为什么要进行 DNS 解析？ DNS 是互联网的一项服务。它作为将域名和 IP 地址相互映射的一个分布式数据库，能够让人更方便地访问互联网。

当 DNS 客户端需要查询程序中使用的名称时，它会查询 DNS 服务器来解析该名称。有时，客户端也可使用从先前的查询获得的缓存信息在本地应答查询。DNS 服务器可使用其自身的资源记录信息缓存来应答查询。DNS 服务器也可代表请求客户端查询或联系其他 DNS 服务器，以便完全解析该名称，并随后将应答返回至客户端。这个过程称为递归。

另外，客户端自己也可尝试联系其他的 DNS 服务器来解析名称。当客户端执行此操作时，它会根据来自服务器的参考答案，使用其他的独立查询。这个过程称为迭代。

因为国内网络环境比较复杂，用户可能来自电信、联通、移动、教育网等网络，所以建议把域名的域名服务器设置为国内的智能 DNS 提供商，比如 DNSPod。

DNSPod 除了可以根据用户 IP 来给出最佳的 IP 以外，还有一些额外的功能，比如网站监控等增值服务。

云服务器

如果用户量没那么大，您也许不需要用到 VPS。比如做一些个人项目，VPS 总是有它的廉价替代品。

一种情况是架设自己的博客或者作品集网站，可以用静态页面实现。首先使用 Jekyll 编译出静态页面，然后托管在 GitHub Pages 或者一些支持静态文件的云服务器，最后通过免费绑定域名，让网站直接上线，方便又快捷。

还有一些针对专门语言和软件的云服务商。比如，如果希望能运行一个自己写的 Node.js 服务，可以把代码发布到 Heroku 等第三方代码平台，就能直接看到线上效果。Heroku 甚至可以方便地加载 MongoDB 插件，来与三方 mongodb 数据库对接，立等可取。云服务商按用量收费价格也比较低廉，并且它们的服务在用户量不大的情况下是免费的。

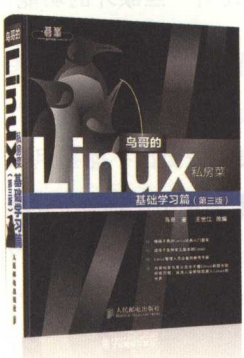
如果需要架设静态资源库，可以放在国内的云储存商那里，比如七牛云存储，除了稳定的服务还有 CDN 的支持。如果要架设 API，那么速度快、价格低的 SAE 是不错的选择。

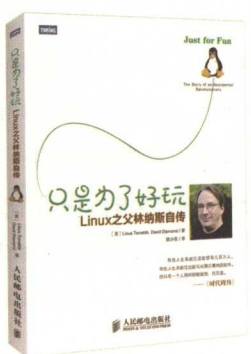
从价格、易用性、功能和扩展性角度看，VPS 总有它的替代品，而且越来越火热的云服务提供商推出了各种各样的服务，现在的全栈工程师可以越来越方便地架设一个网站。

延伸阅读

1. 《鸟哥的 Linux 私房菜 . 基础学习篇》鸟哥，人民邮电出版社

2. 《只是为了好玩：Linux 之父林纳斯自传》（美）Linus Torvalds、David Diamond，人民邮电出版社





软件设计方法

如何学习一门编程语言？很多教科书都会从一个简单的例子开始，那就是输出“Hello World！”，接下来学习一些简单的逻辑、条件判断以及控制等。学完这些通用的基础知识后，进而学习语言的一些特点和特殊技巧，以及标准库的用法。

以上这些比较简单，有经验的开发者几个小时就能学会。但是要真正把代码组织成可运行的系统，而且重要的是，系统代码要可维护、易维护，方便多人协作，这就要求我们学习一致的软件设计方法，也叫做“软件设计最佳实践”。

设计模式

架构模式

设计原则

设计模式

设计模式（design pattern）是从建筑学中引入到软件工程的一个词。设计模式不是一个代码包或者外部库，而是对特定解决方案或者模板的一种描述。设计模式不局限于某种特定语言或者框架，它可以用各种语言和方式来实现。

有的时候设计一款软件或者系统中会遇到一些通用问题，工程师把通用问题解决方案叫做“最佳实践”。设计模式可以看做是精炼、正式的最佳实践。

设计模式的关注点在于以下几方面。

- 高效编写代码
- 高可复用性
- 抽象带来的可读性

在设计模式领域有一本重量级的经典著作——《设计模式》，这本书的作者均是国际公认的面向对象软件领域的专家。[1] 在这本书中，作者列出了 23 种软件设计模式，分为三大类：创建型模式、结构型模式、行为模式。

创建型模式

创建型模式，就是用来创建对象的模式，它对实例化的过程进行了抽象。创建型模式帮助一个系统独立于如何创建、组合和表示它的那些对象。也可以理解为，创建型模式将创建对象的过程进行了封装，作为客户程序仅仅需要去使用对象，而不再关心创建对象过程中的逻辑。

比如“单例模式”（Singleton），就是一种常见的创建型模式。

1 Erich Gamma 博士是瑞士苏黎士国际面向对象技术软件中心的技术主管。Richard Helm 博士是澳大利亚悉尼 IBM 顾问集团公司面向对象技术公司的成员。Ralph Johnson 博士是 Urbana-Champaign 伊利诺大学计算机科学系成员。John Vlissides 博士是位于纽约 Hawthorne 的 IBN 托马斯 J. 沃森研究中心的研究人员。

许多时候整个系统只需要拥有一个全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。

实现单例模式的思路是：一个类能返回对象一个引用（永远是同一个）和一个获得该实例的方法（必须是静态方法，通常使用 getInstance 这个名称）。当调用这个方法时，如果类持有的引用不为空就返回这个引用；如果类持有的引用为空，就创建该类的实例，并将实例的引用赋予该类保持的引用。我们来看看 Java 代码的实现过程。

```
public class Singleton {
      private final static Singleton INSTANCE = new Singleton();
      // 私有构造函数
      private Singleton() {}
      // 默认的公共函数，用来获得该实例
      public static Singleton getInstance() {
          return INSTANCE;
      }
    }
```

可以看到，我们还将该类的构造函数定义为私有方法，这样外部的代码就无法通过调用该类的构造函数来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。

JavaScript 是没有类的语言，它只有对象，但是也可以应用单例模式。一个常见的例子是，当用户点击网址的“登录”按钮时，会弹出一个输入框，同时背景出现一个半透明的黑色遮罩。生成黑色遮罩的代码比较简单，如下所示。

```
$( 'button' ).click( function(){
    var mask  = function(){
    return document.body.appendChild(document.createElement(div));
  }
    mask.show();
});
```

这种方式有一个问题，每次点击按钮时，都会创建一个全新的 DIV 节点（也就是黑色遮罩）。我们如果希望整个页面中最多只有一个半透明遮罩节点，就需要每次都删掉这个节点，显然不合理。所以，更好的方法是使用单例的设计思想，在每次创建之前先检测是否已经存在遮罩节点，如果存在，就直接使用它，如果不存在，就创建一个。

```
var createMask = function(){
    var mask;
    return function(){
        return mask || (mask = document.body.appendChild
        (document.createElement('div'));
    }
}();
```

这段代码首先定义了一个匿名函数 createMask，然后马上执行。

在这个匿名函数的运行环境内，首先创建一个 mask 局部变量，然后返回一个匿名函数，作为“构造函数”。构造函数会判断 mask 变量是否有被初始化值。如果 mask 的值未被初始化，则创建一个 div 节点，并将 div 节点赋值给 mask。如果 mask 存在，就直接返回 mask。

这里还用到了 JavaScript 的“短路运算”的特征。所谓短路运算，是指 ||（或运算符）运算符检查第一个表达式是否返回“true”，如果是“true”，则不再检查第二个表达式的内容。在这里，如果第一个表达式 mask 是已经被初始化的，第二个赋值表达式也就不会被运行。类似的还有 &&（与运算符）。

第一次运行 createMask 时，因为 mask 未初始化，会创建一个新的 div 节点。第二次、第三次……运行 createMask 时，mask 已经被初始化，所以都会使用最开始创建的 div 节点。这实现了我们想要的效果。

除了单例模式，这个例子中还用到了“惰性初始化模式”：推迟对象的创建、数据的计算等是需要耗费较多资源的操作，只有在第一次访问的时候才执行。因为 DOM 操作是浏览器中消耗比较大的操作，所以延迟背景蒙层 DOM 的生成能够提高页面性能。

其他创建型设计模式还包括：工厂方法、抽象工厂、建造模型、原型模式、对象池模式和多例模式等。

结构型模式

结构型模式主要解决类、对象、模块之间的耦合关系。

举例来说，适配器模式（Adapter）是一种常见的结构型模式，它将一个类的接口转换成用户希望的另一个接口。配适器模式使得原本由于接口不兼容而不能一起工作的类可以一起工作。它在多人协作或者兼容旧的 API 时非常有用。

在复杂组件的系统中适配器模式的应用场景很多，下页举一个前端的场景作为例子。

对于一个 DOM 元素，我们有的时候需要设置它的半透明值。如果用原生 JavaScript 代码来实现，代码如下：

```
img.style.opacity = 0.5; // 大部分浏览器，IE9
img.style.filter = "alpha(opacity=50)"; //IE6、7、8
```

因为 IE8 以下版本的浏览器和现代浏览器的实现方式不一样，所以需要在 JavaScript 中使用逻辑判断。

```
if ('opacity' in anchor.style) {
    anchor.style.opacity = 0.5;
} else {
    anchor.style.filter = "alpha(opacity=" + (0.5 * 100) + ")";
}
```

如果我们需要读取某个元素的半透明值，同样需要处理不同的浏览器接口。

看似非常简单的操作，却异常麻烦，对不对？好消息是，通过 jQuery 可以方便地使用同一个方法来读取和设置 CSS 值。

```
$('.elem').css('opacity');
$('.elem').css({'opacity': '0.5'});
```

赞！通过对传入参数的适配，实现不同的功能。如果忽略参数，调用的函数名是完全一样的。通过“重载[1]”CSS 方法，就可以通过参数来区分属性，并且第二个参数是可选的，通过校验参数的数量，来确定用户要实现哪一种功能。

jQuery 里面有很多适配器模式的例子。适配器不会去改变实现层，那不是它的职责范围，它干涉了抽象的过程。适配器只针对外部接口做适配，能够让同一个方法适用于多种系统。

如果内部的实现出现了问题，需要动手术解决的话，就不应该使用适配器了，因为那只是治标不治本的方法，反而会增加代码的复杂度。对实现进行全面优化带来的是真正的改善。而如果实现层的问题不大，要解决一部分适配问题的话，适配器模式就是很好的选择了。

其他的结构型模式还包括桥接模式、组合模式、装饰模式、外观模式、享元模式和代理模式等。

行为型模式

行为型模式为设计模式的最后一种类型，用来识别对象之间的常用交流模式并加以实现。如此，可在进行这些交流活动时增强弹性。

比如，观察者模式（Observer）是一个具有代表性的行为型模式。在此种模式中，一个目标对象管理着所有依赖它的观察者对象，并且在它本身状态改变时主动发出通知。

看起来好像很抽象，下面我举一个实际的例子。我们去餐厅吃饭，人比较多时需要排队，我们不用隔两分钟就问服务员“轮到我了吗”，而是坐在那里等服务员的通知。大家都等着叫号，叫到谁，谁就去。服务员就是“被观察者”，也就是一个主题，每位顾客就是一个“观察者”。因而，观察者

1 事实上，JavaScript 并不是完全支持“重载”，因为它只能定义一个函数签名（比如 CSS 函数），然后通过参数的个数来进行逻辑区分和判断，实现类似“重载”的效果。

模式也叫做发布订阅模式。

在 MVC 的架构模式中，观察者模式很常用。因为我们希望数据（model）的变更可以自动被界面（View）观察到，而不是依赖数据对每一个界面都进行主动通知。这可以减小前后台开发中的耦合性。

其他行为型模式还包括黑板、责任链、命令、解释器、迭代器、中介者、备忘录、空对象、模板方法和访问者等。

架构模式

架构模式专指用来解决项目架构问题的模式。

有些人认为，架构模式是设计模式的一个子集；也有些人认为，架构模式跟设计模式是并列关系，都是为了教我们如何高效地写代码。

其实并没有那么严格的定义，当我们自上而下设计系统的时候，可以认为 MVC 是一个抽象程度很高的设计模式。

架构模式教我们如何架构一个系统，它的关注点在于以下两点。

- 多个职位（比如后台开发和前端开发）可以平行工作同时进行。
- 构建一个软件系统的多种技术。

MVC 模式

MVC 模式（Model-View-Controller）是最有名的一种架构模式，由于它在各种系统中被广泛使用，有些人称它为“架构模式之王”。

MVC 把软件系统分为 3 个基本部分：模型（Model）、视图（View）和控制器（Controller），每个部分放在不同的地方维护，绝不互相干扰。MVC 被证明是一种非常有效的提高系统架构的模式，以至于有一些程序语言在

发布初始不温不火，但是其他人开发了一个这种语言的 MVC 框架之后，就迅速爆红，风头甚至盖过语言本身。

说到代码的可维护，其实并不只是为了方便他人，也是为了方便未来的自己。根据大量的统计和观察，其实工程师在编程的时候花费时间最多的不是敲键盘，而是读已有的项目代码、分析代码架构和已有的逻辑。读自己一年前写的代码，与读其他人的代码其实没什么区别。

工程师水平高低并不在于敲代码有多快、复杂性多么高（当然这是高智商的一种体现），而是能快速理解其他工程师的代码，并且自己编写的代码也能让其他工程师快速理解。

这也是现代大型组织运行所要求的一种方式：每一个岗位都是可配置、可插拔的，而不是某一个工程师写出的程序其他人都理解不了，或者某一个销售人员掌控公司大部分客户资源。如果是这样，公司的风险会很大。但是，现代大型组织非常复杂，如何能让每个新员工都能快速加入，而且公司的运作可以掌控？答案是划分不同的部门和岗位，让每个人的工作简单化。

软件系统也是如此，现代软件非常复杂，但是可以通过分解的架构方式来减少每个功能模块的耦合性。

Rails 是一个使用 Ruby 语言编写的开源 Web 应用框架，是严格按照 MVC 结构开发。它努力使自身保持简单，来使实际的应用开发代码更少，使用最少的配置。Ruby 背后的设计思想是方便工程师优雅快速地编码，但是 Ruby 发布之后，并没有引爆 Web 开发。后来，是 Rails 的发布真正让 Ruby 火了起来，以至于现在（在 Web 开发的语境下）提到 Ruby，都是指 Rails。

架构模式之王

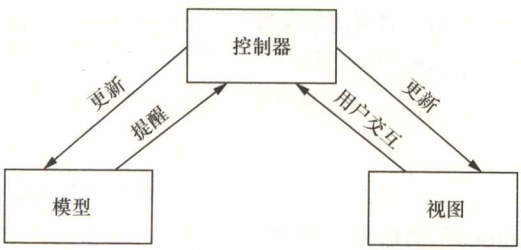
MVC 的本质是代码的分离，它不光在 Web 开发中大放异彩，在桌面软件

和移动端软件开发中也随处可见它的身影。所有需要多人协作的项目都可以使用 MVC 来架构，而所有的项目最终都会需要多人协作。

以 iOS 开发为例，当您新建一个 Single View Application 的时候，它并没有强制您一定要使用 MVC 的方式去编码，您大可以在一个 ViewController 中完成所有的工作。但是为了良好的架构风格，我们最好创建 3 个组，分别叫 Controller、Model 和 View，然后每次创建一个类的时候，都确保这个类属于三者之一。

在命名上，以一个相册为例，model class 的命名就叫 Album.h 和 Album.m；view class 的命名叫 AlbumView；controller class 叫 AlbumController。这样就能很好地区分 MVC 的类和实例。

在 iOS 的 MVC 架构的通信模型中，Controller 属于驱动整个系统运行的核心，俨然发挥着心脏的作用，所有的数据都要经过 Controller 才能传递给 Model 和 View：



MVC 架构模式。

通过这样的架构方式，我们在修改视图的时候，就不会受到数据的影响。如果有一些页面共享视图模型，但是数据来源不一样，就更简单了，只需要调用不同的数据模型即可。

MVC 作为设计模式之王，有一些设计思想和设计模式围绕着它。理解这些设计模式的背景和原理是非常有趣的。但是光谈理论是没用的，一定要多动手去写，有些设计模式可能您要走弯路才能发掘它的必要性。有一个学习方法就是，在目前在用的语言后面加上 mvc framework 去搜索，比如：

- JavaScript mvc framework: AngularJS、backbone.js
- Php mvc framework: CakePHP、Yii、Codeigniter
- Objective-c mvc framework: Cocoa
- Ruby mvc framework: Ruby on Rails
- Nodejs mvc framework: Sail.js、Total js、Express

设计原则

Rails 使用了 MVC 的架构模式，以及大量设计模式（包括上文提到的单例、适配者、观察者等），但这还不是全部，它还使用了一些设计原则。设计原则更像是大的指导思想，就像数学的公理，基于公理可以推导出很多推论。

在 Rails 设计之初，作者就设定了几条设计原则，包括“不做重复的事”（Don't Repeat Yourself）和“惯例优于设置”（Convention Over Configuration），下面我们来分别介绍。

DRY

DRY 是“Don't Repeat Yourself”的缩写。意思是说，在一个系统里，对于任何数据或者变量，都应该配置在有且只有一个地方，其他的地方都应该引用这里的数据。这样，需要改动数据的时候，只需调整这一处，所有的地方就都变更过来了。

DRY 是一个非常入门级的设计原则。从小处讲，我们在创建每一个变量的时候都有这个设计模式在背后支撑，即避免在所有地方都使用常量。在高一点的抽象层级上，代码功能的细化和分离都是在为 DRY 做支撑——让我们的代码模块可以复用。

DRY 跟系统架构的关系很密切，在有些场合下即使是资深工程师也会受限

于当前的项目架构，而写出重复的代码。反之，如果有良好的项目架构，即使是初级选手也能按图索骥，写出良好的 DRY 代码。

比如，在前端项目的代码库里面，每次新建一个页面时，都复制一份重置样式[1]，那就是很糟糕的方式，因为要修改重置样式的时候，需要同时修改所有的样式。但是如果项目代码库是扁平层级的，所有文件都是以组件的方式相互依赖，每个组件都是唯一的，并且通过工具合并成唯一的文件发布到外网，那么这个架构就会强制开发者引用已有的文件，而不是复制出一份代码。这样的好处是，真正需要修改全局样式的时候，修改和发布变得非常方便。这就是好的架构带来的优势。

那么是不是所有需要重复的地方，都单独提取出一个模块或者组件呢？不尽然。三次法则（rule of three）是代码重构的一条经验法则，涉及当代码片段出现重复时，如何决定是否用一个新的子程序替代它。三次法则的要求是，允许按需直接复制粘贴代码一次，但如果相同的代码片段重复出现三次以上，就需要将其提取出来做成一个子程序。

DRY 的反义词是 WET（Write Everything Twice，所有的东西都写两次），“干”跟“湿”对应，很有意思。

惯例优于设置

Rails 的一个主要设计思想是“惯例优于设置”，它的本质是说，开发人员仅需规定应用中不符约定的部分。例如，如果模型中有个名为 Sale 的类，那么数据库中对应的表就会默认命名为 sales。只有在偏离这一约定时，例如将该表命名为 "products_sold"，才需要写有关这个名字的配置。

如果您所用工具的约定与您的期待相符，便可省去配置；反之，可以修改配置，来达到所期待的方式。

---

1 Reset CSS。不同浏览器的默认样式往往不一样，会给我们带来麻烦，影响开发效率。所以解决的方法就是一开始就将浏览器的默认样式全部去掉，更准确地说就是通过重新定义标签样式来覆盖默认样式。

PHP 是一门专门为 Web 开发而发明的语言，它也有一些 PHP 的开发框架，我比较熟悉的就是 Codeigniter，它非常简单，学习曲线也很平滑。我曾经用一周的时间学习 Codeigniter，并编写了一个开源系统，它登上了 GitHub 热榜。这个项目现在还受到 Codeigniter 中国官网的首页推荐，因为它就符合“惯例优于配置”这一设计原则。

Codeigniter 的安装和部署非常方便，除了数据库连接方面的一些必要配置，整站都无需配置。比如当我需要一个路由 `http://siteurl/blog/` 页面的时候，我只需要在 `application/controllers/` 文件夹里创建一个 `blog.php`：

```
<?php
class Blog extends CI_Controller{
    function construct() {
      parent::construct();
    }
    public function index() {
      echo 'Hello World！';
    }
} ?>
```

然后访问 `http://siteurl/blog/` 的时候，就可以看到页面上输出了“Hello World！”。

如果我需要在 URL 中有更多的方法，比如 `http://siteurl/post/123`，只需要为 `post` 类增加一个方法即可。

这就是“惯例优于配置”的一个典型例子，它让我的工作更简单，当需要修改 URL 的时候，更不容易出错。

在 Node.js 的 MVC 框架 Express 中，路由就需要有一个单独的配置的地方 `app.get('/', routes.index);`。我觉得这样增加了复杂性，因为要同时修改路由配置和 Controller 的文件名。

KISS 原则

KISS 是“Keep it simple, stupid”的缩写，意思是说软件设计当中应该注

重简约的原则。这一原则认为，大部分系统的设计越简单越好，有不必要的复杂性都应该避免。如果一个系统非常复杂，就应该分解为多个简单的组件，做好足够的分解和抽象。

在用户界面设计中也常常参考这一原则，更简洁的界面设计，能让所有人看得懂，不要添加太多花里胡哨的功能。

有一次，我们的设计师设计了一个注册页面，用户需要填十几项表单才可以注册。我就建议，注册页面要尽量简单化，我们只要得到尽量少的、足够让系统工作的信息就可以了，更多的信息可以让用户在注册成功后自己补齐。或者在需要的时候再提醒用户补齐。注册时太多的选项可能会让用户放弃注册。

简单系统或方法的好处在于以下几点：

- 较简单的系统更容易构造、运行和维护。
- 较简单的解决方法总是更具弹性、柔性。
- 较简单的系统更便宜。
- 较简单的系统更容易实现、更快地获得回报。
- 较简单的方法更讨用户的欢心。
- 较简单的系统更容易分阶段地执行。
- 较简单的系统更容易被理解。

最少知道原则

最少知道原则[1]，常常用在面向对象软件设计中，它是“松耦合原则”的一个具体实例，是指在面向对象编程中，每一个软件单元应该尽可能少地与其

1 最少知道原则（Least Knowledge Principle），又称为迪米特法则（Law of Demeter）。

他单元发生作用。

最少知道原则又常常被这样描述：

> 每一个单元都应该只知道关于其他单元的有限信息：那些与自己密切相关的单元。
> 每一个单元应该和自己的朋友讲话；不跟陌生人讲话。

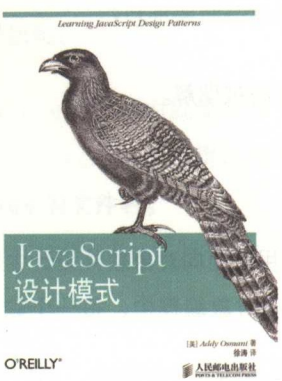
最少知道原则的理论基础是，如果一个单元与其他单元的联系太紧密，那么这个单元需要深刻地理解其他单元的信息。在这种情况下，如果其他单元发生了修改，那么所有与之密切相关的单元都需要进行仔细的排查，然后做出对应的修改。因为没人知道这个单元跟其他单元有哪些紧密的联系。

应用最少知道原则之后，软件系统能够得到更高的可维护性。因为对象都只暴露自己少数对外的 API，而隐藏了自己的内部结构和实现原理。所以如果一个类需要重构的时候，只需要保持 API 不变，这样就不会影响整个系统的运作。

延伸阅读

1. 《设计模式：可复用面向对象软件的基础》（美）Erich Gamma / Richard Helm / Ralph Johnson / John Vlissides，机械工业出版社

2. 《JavaScript 设计模式》（美）Addy Osmani，人民邮电出版社



高效工程师

在相当长的一段时间里，Facebook 的格言都是："Move fast and break things"（快速行动，打破常规）。在 2014 年的 F8 开发者大会上，扎克伯格将这句话改成"Move fast, with Stable infra（快速行动，稳定架构）"。

在过去十年中，Facebook 获得了巨大的变化和发展，这与它的企业文化是分不开的。但是，Facebook 现在除了在意速度，还更注重为用户提供稳定的服务。

"我们的这句格言曾经很有名……背后的主要想法是，作为开发者，快速行动是如此重要，以至于我们能够容忍有一些 bug。但是现在我们意识到，如果我们总是不得不停下来修改这些 bug 的话，那我们不可能快起来。"扎克伯格说。

虽然现在的格言没有之前那么特立独行，稳妥了许多，但是我们可以看到，Facebook 对于速度的追求一直没有变。

为什么需要高效

提速 100 倍

为什么需要高效

互联网公司对高效能工程师的需求很迫切。

过去，一个机构的组成多以体力劳动者为主体，如操作机器的工人，或者前线打仗的士兵。这样的机构组成，对高效的需要不太迫切，不高效导致的问题也没有今天严重。

关于体力工作，我们已有一套完整的衡量方法和制度，从工作时间、工作产出到工作质量，都有客观的衡量方法，但是这种衡量方法和制度并不适用于知识工作。通过产能评估，我们可以清楚地评估一个工人的效率是另一个工人的多少倍，但是对于知识工作，我们无法评估效率。**知识工作者并不生产具有效用的产品，他生产的是知识、创意和信息。**

而且，一个偷懒或者低效的工人也不会比一个高效的工人差太多，二者生产力的差别最多可能是两倍。但是，高效的工程师的生产力是低效工程师的几十倍。

在过去，一个工人如果干得比较差，他最多只影响自己，而不会影响他人。但是今天的工程师需要跟上下游共同编写程序，生产他人所需的输出。一个人的效能会影响整个团队的效能，所以每个人的高效都很重要。

最后，高效的工程师能够拿更多时间运动和休息，这对于长期稳定的工作投入是有帮助的。低效工程师长期熬夜带来的副作用完全抵消了工作时长上的投入。

提速 100 倍

如果您从心底认为高效比加班更有效，我们来看看怎样提高效率。

阅读英文资料

对于全栈工程师来说，遇到问题时常常会求助搜索引擎，尤其是在学习新

的编程语言，或者学习某个框架的新版本的时候。这是最快速的方案，在很多时候比看官方文档还有效。有的时候我们只是想快速了解一个新的技术，或者快速上手，这时候也会用到搜索引擎。

为什么我推荐使用英文，而不是使用中文搜索，有这样几个原因。

- **英文的技术资料更多**

在全世界范围内，IT 工程师普遍使用英文（即使中国工程师的数量也许更多），于是用英语写作就有了最大的读者群，然后有更多的人用英语写作……。

在 Google 中用英文搜索技术问题，往往能得到很详细的说明，以及相应的代码示例。有的时候您会进入英文维基百科，里面有关于词条的详细解释，包括历史、版本、理念、竞争者，等等。有的时候您会进入一个教程（tutorial），里面会详细介绍完成一个任务的整个流程，甚至还可能贴心地提供工程示例文件的下载，这是新手学习的最好方法。

- **StackOverflow 有完善的鼓励机制**

内容筛选机制、严格的内容把关政策，这让 Google 的内容质量很高。如果您键入的关键词足够多，那么您很有可能看到很多 StackOverflow 的问答链接。

StackOverflow 是一个与计算机编程相关的问答网站，也是最大的 IT 工程师平台。在这里有很多知名开发者，由于有完善的系统鼓励政策，他们都很热衷于回答其他开发者遇到的问题。用户可以在网站免费提交问题，浏览答案，赞同或者反对答案。StackOverflow 是一个英文网站，所有用户都必须用英文提问和回答问题。很多时候，在一个问题下都会有多个回答，有经验的用户可以赞同或者反对其他人的回答，新手可以根据每个回答的赞同数来判断回答的可靠性。

中文互联网社区中也有很多专注优质文章的博客，比如我很喜欢的阮一峰

博客，他十几年始终如一地学习新技术，并且把学习过程和心得记录成博客，发表频率和质量都非常高。但是这样一些个人的努力并没有由一个社区聚合起来。有一些做得不错的中文社区，比如 V2EX，从它的一些机制上也能看出创始人对社区理念的重视，希望以后能有更多多样化的社区出现。

关于 StackOverflow，还有一点题外话。StackOverflow 由两个世界著名的博客作者共同创建，Jeff Atwood 和 Joel Spolsky。Jeff Atwood 提出了著名的 Atwood 定律："任何能够用 JavaScript 实现的应用系统，最终都必将用 JavaScript 实现。" Joel Spolsky 可能是软件工程师中写博客最有名的一位，他是 "Joel on Software" 博客的作者，读者人数可以排进全世界前 100 名。两位创始人对工程师心理有着深刻的洞察，包括追求荣誉感这一特点，所以他们设计的 StackOverflow 的鼓励机制，取得了巨大的成功。

- **Google 的搜索能力非常强**

Google 在搜索引擎方面的技术实力可以说是绝对领先，它比排名第二的搜索引擎的技术实力和用户体验要强一大截。我曾经对比了大量词汇在不同搜索引擎中的搜索结果，有时候差不多，有时候差别很大，Google 总是好得多。所以即使您要搜索中文问题和词汇，也仍然建议使用 Google。

- **英语世界的语言风格比较严谨**

中文互联网世界上的语法一般比较口语化，而且不太在意语法正确性和逻辑的严谨性。在整篇文章的行文和逻辑构建上，英文也往往更严谨。

如果您的英文不够好，没有关系。在长篇的英文技术文章或者 StackOverflow 中，写作者都会有意使用最简单的词汇和语法。如果您能编程，您就能理解这些简单的英文。

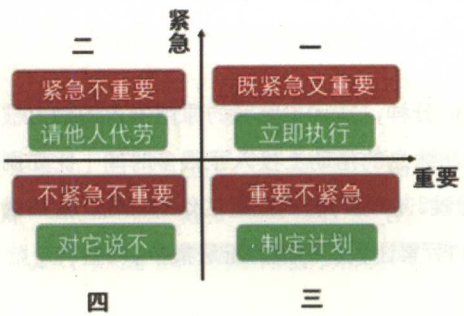
时间管理四象限

在我刚进入腾讯工作的时候，一个领导就跟我说过，**如果您平时没有做重**

要的事情，就会发现自己常常在做紧急的事情。如果您平时没有注意锻炼身体，就会常常去医院，花费更多时间。如果没有培养后辈，为每个项目设置接班人，就会常常需要到处救火。如果您没有配置好版本管理系统就开始工作，就会浪费更多时间去找回丢失的代码。如果偷懒硬编码（hard code）了一些变量在代码中，后续一定会花费更多时间去调试。

时间管理四象限将我们平时需要做的事情分为 4 类：重要而且紧急、重要但不紧急、不重要但紧急、不重要而且不紧急。

时间四象限教我们把不同的事情分在不同象限之中，然后采用不同的处理方法。



时间管理四象限。

第一象限是重要又紧迫的事情。比如线上出现严重 bug、服务器出现安全漏洞、用户的投诉，等等。对于这些问题，是考验自己的经验和判断力的时候，用自己最好的判断力，立即执行。不过也要意识到，在这一象限的问题往往是第二象限的问题没有制定好的计划而滑落过来的，如果大部分的工作都落在第一象限，说明还是在“瞎忙”。

第二象限是紧急但不重要的事情。比如会议、一些可转交的需求，等等。这种事情往往会由于对方强烈的呼声让您认为它处于第一象限，有一种“这事很重要”的错觉。我们花很多时间在这里打转，其实只是在满足他人的需求，而没有关注自己的职责。对于这种事情应该请他人代劳，不一定是

自己的下属，还可以是平级的同事。有一些会议不一定需要自己去参加的，可以提前跟主持人（项目经理）沟通。

第三象限是重要但不紧急的事情。比如一些线上体验优化计划、团队的长期发展方向和个人培训等。有一次老板提出了线上的一个体验问题，我们以为这是第一象限的问题，马上执行。结果老板说，这个事情不需要您们这么快去执行，希望可以做深入一点的调研。**建议工程师把 80% 的工作投入到该象限中，避免“瞎忙”。**

第四象限是不重要也不紧急的事情。逛论坛、刷朋友圈就属于这一象限。简而言之就是浪费生命，所以根本不值得花半点时间在这个象限。但我们往往在一、二象限来回奔走，忙得焦头烂额，不得不到第四象限去疗养一番再出发。所以我们要避免浪费时间在第四象限，也不要疲于奔命，各处“救火”。

有时候我们以为是外部原因让我们分神，其实更重要的原因是人的精力被耗尽了。我们在一些以为是消遣和休息的活动上投入了很多时间（比如刷朋友圈、玩游戏），回到工作的时候，并没有感到重返工作的舒畅，反而精力涣散，容易分神。我们身体上的劳累让我们的精力无法集中。

消除重复工作

重复的工作应该交给计算机去干，这是我们都知道的一个道理。

有趣的是，我们往往不知道我们的时间花在哪里，有一个办法就是详细地记录自己一天的时间消耗。在合并代码上面花费太多时间？还是提交测试？发布流程繁琐？编译太久？切图的工作枯燥无味？

第一步便是识别出自己的时间花费在了哪里，以此作为优化的目标。

有了优化目标之后，第二个思考的问题是，能否使用已有的工具——免费的或付费的——来无缝衔接在已有的流程中。如果新的流程成本太高，那就无法有效推进。普通工具可以用英文在 GitHub 上搜索相关的关键词，前

端开发者可以在 Gulp 和 Grunt 社区寻求帮助。

如果不能使用已有的工具的话，就要自己去编写了。在语言的选择上可以从几个角度考虑：工具的使用者是谁？如果需要分发给同事安装，那么他们使用什么系统？Windows 还是 OSX？如果使用 Web 来作为用户接口，那么就可以使用任何您喜欢的语言。用脚本语言开发小型工具非常快，无需编译即可运行。Paul Graham 在《黑客与画家》里面就提到过他使用 Lisp 语言来快速开发的经历。

给自己留出不被打扰的时间

当我们被 QQ、微信、RTX、电话等提示工具打扰时就会频繁分神，这使得专注力如今已成为稀缺资源。

Paul Graham 在一篇文章中说过，创作者的时间计划与经理的时间计划是非常不一样的。工程师作为创作者，遵循创作者的时间计划。**编写程序需要大量的精神投入，需要整块连续的时间思考**，所以工程师工作时是不希望被打扰的，若思路被打断则后果很严重。

Facebook 的每周三是没有会议的日子（No Meeting Wednesdays），每一个团队成员都知道，除非万不得已，不要在这一天安排任何会议。事实证明这对团队的效率很有好处，它鼓励所有人（包括产品经理）成为“做事情的人”而不是“计划事情的人”，而且所有人都有一整块的时间工作。对于远程工程师来说，还可以选择在家里工作，根本不用出现在办公室。

番茄工作法

有些时候，我们没有办法得到一整天的大块、连续的工作时间，为此可以采用短迭代的工作方法，高效地利用每一小块的工作时间。

我在跟 Facebook 的工程师聊天的时候，发现他们都很注重锻炼身体，每天早上开始工作之前都会花一个小时跑步和健身，然后洗澡吃早餐，再开始一

小段高效的工作时间。下午的工作时间中也会穿插一些运动和甜点时间，等等，不会像国内的工程师们，需要长时间地坐在电脑前。这种状态的转换让他们的效率得到了很大的提升。所以包括腾讯在内的一些国内公司现在也开始把健身房和团队运动作为一项员工福利，有时候还会组织晨跑或者徒步等团体运动。

“番茄工作法”是由弗朗西斯科－西里洛于 1992 年创立的一种微观时间管理方法。使用番茄工作法，选择一个待完成的任务，将番茄时间设为 25 分钟，专注工作，中途不允许做任何与该任务无关的事，直到番茄时钟响起，然后在纸上画一个 X，短暂休息一下（5 分钟就行），每 4 个番茄时段则多休息一会儿。

番茄工作法极大地提高了工作效率，还会有意想不到的成就感。为什么 25 分钟是一个比较好的时间点？因为如果过短，思维还没有恢复过来，就马上要被打断，不利于创作。番茄工作法主张在 25 分钟时间段内专注地完成高质量的工作，接着是 5 分钟的休息。如果让压力系统一直工作，而不借助心智休闲进行调节，一些症状会找上门来。

跨界思考

有时候项目会频繁变动，从 A 计划临时转变成 B 计划，然后再回到 A 计划的事情屡见不鲜，这是浪费工程师精力的最大因素。

从项目的角度来讲，也许这是好事情，谁也不希望把不好的产品发布出去。每个人都希望自己做的产品能够成功，但是不是做重复的脑力劳动。归根结底，是双方在沟通的过程中充满了障碍，不同岗位的人的思考问题的方式不同。

设计师们是典型的浪漫主义者，他们的世界里没有公式、没有定理、永远没有标准答案，更多的是用发散的思维去构思创意，更容易产生好想法。而工程师注重逻辑的严谨性、技术的可行性，他们往往用“我是否能够做到”和“我要花多少时间才能做到”这个思维在思考。

那么这种矛盾如何解决？在初级设计师和初级开发者之间，我们鼓励他们频繁沟通，把自己的想法用对方能懂的语言说出来。在高级设计师和高级开发者之间，界限就没有那么明显了，我鼓励他们帮对方做一些工作。这说明什么呢，沟通固然重要，跨界学习更加重要！当双方带着完全不同的思路去讨论一个话题，讨论出来的结果必然是双方妥协的产物，而不会是最优的结果。

很多工程师都向往国外大公司的“工程师文化”。为什么？因为“工程师文化”代表了工程师是产品开发中的强势方，在跟产品和设计师的沟通中有更多的话语权（甚至某些产品跳过设计师和产品经理）。但是我认为，这是**因为国外的工程师有更高的跨界思考能力，才得到了自己的投票权**。很多工程师主导的项目，界面不华丽，但是可用性都非常高。这就是跨界能力带来的另一个好处，可以做一个“产品创作者”。

过去跨界学习的成本很高，大部分人都不敢轻易尝试。但如今互联网时代给我们带来了机遇，每天上网都可以看到其他领域名人写的文章和微博，通过查看这些内容，我们就能对原本完全陌生的领域有一个感性的认识。时间一久，我们就能够在潜移默化中理解另一个领域的从业者的思维方式，当您开始跨界学习之后，就会增加更多的机会。

我在工作的这几年中也在渐渐受到设计师和管理者的影响，开始学习设计师和管理者的思维方式，所以有时候我被认为是“有一点设计感觉的工程师”，但我仍在学习。**或许每个工程师会在不同的环境中跨不同的界，但是在未来，我认为跨界出来的那部分能力才真正定义了“您”。**

纸上头脑风暴

我使用过很多工具，比如 MindMap 类软件，或者简单的记事本软件、手机上的绘图软件等。最终，我的经验是，在电脑上工作之前，先在纸上画出自己的想法。**笔跟纸是最灵活、最容易修改、成本最低的头脑风暴方式。**

比如在写一篇文章前，就可以先画一个思维导图，把头脑风暴出来的所有

关键词都列出来，然后再根据金字塔式的写作方式层层分解。如果是开发一个软件，或者写一个脚本程序，可以把每一步的主要工作都写下来，类似伪代码，但是抽象层级更高一点。

使用版本控制工具和构建系统

数据表明，对于没有使用版本控制工具的工程师而言，浪费在代码合并、版本回溯以及 bug 修改上的时间远远超过了使用版本控制工具的工程师。所以，这是针对所有人都适用的建议，尽快使用版本控制工具。

好的构建系统可以让工程师完全从枯燥的操作中解脱出来。在极端情况下，不使用构建系统的工程师可能要花费 80% 的时间在编译和等待生效、清除缓存等操作上。

版本控制工具的具体做法已经在《持续集成》这一章中说过了，这里就不再重复了。

加班是一种文化?

我不认同加班是一种“文化”。加班就是加班，是由于工作没有完成导致的。

在一个项目和人力都很稳定的团队中，有两种原因会导致加班。一种是糟糕的项目管理，领导失职，没有安排好工作。第二种是员工能力不够，效率不高，没有按时完成目标。在这两种情况下，我都建议您跟领导去沟通，

我时常听人抱怨说，团队有加班的氛围，他早早完成了工作，可是看见大家下班后还待在座位上（尤其是领导还在），就不好意思自己一个人下班，只好留下来“陪加班”。甚至，因为长期在这种环境下工作，有人认为领导在升职加薪的时候优先考虑爱加班的人，所以白天就慢慢吞吞，晚上就开始加班。

我认为这种情况也许只是个别人的认知问题：认为领导推崇加班的氛围，以及领导按工作时长来判断投入度，以此作为升职加薪的考量因素；或者

认为领导不知道谁工作成果更好，所以按劳累程度来判断……所有这些认知都是因为茫然和不自信，不知道自己的核心竞争力是什么，也不知道老板真正需要的是什么。**拿工作时长来拼，这还是体力劳动时代打工者的心态在作祟。**

稍有常识的老板，或者接受过一点点管理培训的领导者，都知道评估员工是看结果，而不是看努力和过程。您在把老板当傻子，拼命展示“辛苦”，结果更加不如意，只好天天抱怨：“小 A 这个家伙每天不加班，怎么老板那么喜欢他，升得比我还快！”

工程师需要提高自己的沟通能力，如果大家因为紧急项目等原因在加班，您不妨过去问问，能不能帮到什么忙。用积极的态度去思考问题，而不是消极地坐在那里，浪费自己的时间和情绪。大部分老板都会很欢迎员工主动的沟通。

如果团队内年轻人比较多，有些时候他们下班后喜欢留在办公室，并不是因为项目需要，而是公司内有免费的晚餐、空调、饮料、电脑……几乎是一个单身宅男需要的一切了，所以他们喜欢待在公司也是比较容易理解。我常常跟他们说的是，下班之后尽量不要处理工作需求了，多点时间自我学习，或者准备一些分享，甚至做一些编外项目（side project）。重点是，不要加班做白天遗留下来的工作。

如果下属经常需要加班才能完成工作，我会认为是管理者的失职，一方面我会看看是否给员工安排的工作太多，能否转移一些给其他人，另一方面也会看是否需要借调其他资源，或者招聘更多人手。

加班狂的英文是 Workaholism，直译为“工作上瘾”。长期加班的人会有一种错误的观念，就是午夜鏖战全情投入项目的标志。他们不会去找高效的方法，因为他们确实喜欢加班。这往往导致恶性循环，熬夜和过量的工作让人疲惫，影响人的思维和判断力，这样更加无法找到高效的方法来完成工作。

最后，过量的工作和压力严重影响健康，最终导致一些慢性病，甚至猝死。我们经常在媒体上看到某 IT 公司员工猝死，临死前还在朋友圈发加班的照片。

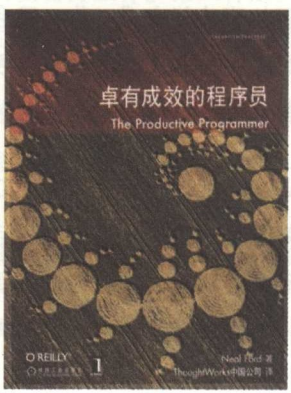
我不确定家庭状况、个人健康或者具体项目对猝死员工产生了什么影响，但我认为一个接受过高学历教育的成年人，应该有能力权衡自己的健康、家庭的责任，以及长期工作回报。每个人都可以有自己的选择。工作上瘾者不光把自己逼上绝路，也把老板逼上绝路。您认为自己能高负荷工作坚持一个月，他就认为您还能坚持半年。

总结一下，加班不是一种“文化”，它往往是我们心中假想的老板，还有媒体渲染出来的一种畸形工作模式。在这种模式中，加班狂习惯了熬夜和过量的工作，他们往往也给了老板长期加班的预期，同时也带来低迷的士气，以及疲倦的身体。

延伸阅读

1. 《软件随想录：程序员部落酋长 Joel 谈软件》（美）Joel Spolsky，人民邮电出版社

2. 《卓有成效的程序员》（美）Neal Ford，机械工业出版社



学习设计

在前面的章节“如何成为全栈工程师”里，我给有志成为全栈工程师的同学的第一个建议是“关注商业目标”，第二个建议是“关注用户体验”。而好的设计是优秀用户体验的必要非充分条件。Web 产品的体验涉及很多参数，比如是否满足用户需求、页面加载速度、响应速度、交互逻辑、稳定性……但是好的设计可能是其中最直观的一个。

在“从学生到工程师”一章里，我讲了自己在面试过程中的经历。虽然我的目标是技术岗位，但是在一半的面试时间中，面试官都让我谈设计。而事实证明，在后来的工作中，设计能力帮了我很大的忙。

我认为工程师不一定要有很强的设计能力，但是对好设计的感觉和判断力则非常必要。所以接下来我想谈谈为什么“设计感”对工程师很重要，以及工程师怎么学设计。

科学家和工程师

设计基础

Facebook 的品牌设计故事

科学家和工程师

首先，一个事实是：**过去的工程师普遍不在意设计。有意或者无意，他们忽视设计的重要性。**

知乎网站上有一个帖子，问题是“为什么部分开发工程师不喜欢调节界面的 UI 细节？”。我比较赞同下面这个回答：

> “我发现程序员大致可以分为科学家和工程师两类，科学家关注技术是否优越，而工程师关注产品是否完美。和科学家类型的程序员合作项目往往是件痛苦的事情，他们太过关注自己手中的锤子是否先进，却不在意自己敲进去的钉子是否平整光滑不扎屁股，更不要说这颗钉子是不是跟其他钉子对齐了。那些“资深”程序员更是如此，那个年代很多用户体验的技术不成熟，能做出一个能用的东西已经不易，更不要说做出一个性能还算不错的产品了。抱着这个想法走到今天，大多数应该被淘汰的程序员却反而坐到了更高的位置，开始拿这种过时的想法熏陶小弟。”（来自陈鑫的回答）

8 年前我刚进入大学的时候，就能感觉到软件教育还停留在理论和技术的阶段。学校的专业包括软件工程、计算机系统等，而没有 Web 设计和软件 UI 这样的专业。在理工科学校，软件 UI 不需要专门的高等教育；美术类高校也没有软件 UI 设计这门课，我们的设计师很多来自平面设计或者工业设计专业。

归根结底，是因为软件开发还处于婴儿时期，尚未进化到对 UI 要求如此高的程度。

随着工业的发展，竞争越来越激烈，好的产品设计成为强大的竞争力，所以诞生了工业设计这一专业（甚至还要更细分到汽车设计和家居设计）。随着用户可以下载和使用的软件越来越多，可挑选的产品越来越多，他们对软件 UI 设计的要求也正在越来越高。

对产品设计要求的提高往往催生了新的岗位划分，很多工程师会说“我就是个开发，界面找设计师去”或“我就是个搞技术的，做没有技术含量的

东西不是我的理想。”但是绝对的细分并不总是最好的解决方案。

细分不是最好的解决方案

工业时代诞生了流水线，它让人类快速高效地生产出大量的商品，这得益于工人的分工，上一个流程完成才开始下一个流程。

产品设计属于最上游的流程，在做实体产品的时候，要用 3D 绘图软件设计一份完全模拟实物的设计稿，然后做一个样品，确定通过后就可以批量生产了。

前一段时间我家里要装修，请了一个设计公司来家里勘测和设计装修方案。设计公司会安排一个设计师上门测量空间大小，然后回去用电脑创建一个房屋的 3D 模型，再根据我的要求给出设计方案，完全符合房屋大小。设计公司在前期确定设计稿方面花了较多的时间，因为设计公司会根据完全确定的设计方案去采购墙纸、瓷砖、木地板和顶灯，这些物件是不可以更换的，一旦确定就不能更改。

实体产品修改设计的成本非常高，所以在设计阶段会精细打磨，确保万无一失。

但是在软件开发中，更细致的分工不一定会更高效。

以响应式开发为例，现代 Web 界面开发最大的一个挑战是，页面可能运行在任何设备的任何浏览器中，Windows 设备、OS X 设备、支持 Retina 屏幕的设备、iPhone、iPad，以及各种 Android 设备，等等。如果说实体产品的形态是确定的，Web 产品就是“不确定”的，那么如何开发适配多种浏览器和屏幕尺寸的页面?

在传统 Web 设计流程中，我们也参考了工业化的流水线：按交互设计、视觉设计、前端开发、后台开发的流程来生产一个产品。按照这种细致的分工，设计师就要输出 3 个以上的页面：手机、平板、超大屏幕的电脑等。设

计师在交付设计稿的时候如果有需要调整的地方，3 个页面就要重新调整。而且对于前端工程师来说，阅读和实现这 3 个设计稿也是很大的挑战。只要设计发生了调整，就要更改每个页面对应的样式，大量的人力和时间就浪费在不必要的“流程”中。

我们采取的优化方案是，视觉设计师关注设计模块和整体氛围，只需要给出一份为 PC 设计的视觉设计稿。让有一定设计理论基础的前端工程师，根据拿到的 PC 设计稿，利用流式布局和媒体查询等技术，直接创建可以适配多个平台的页面，也就是一个“响应式”的页面。

另一个例子是，我们想在页面或者 App 中创建一些动画效果。在老式 Web 设计中，不会有很多的动画，点击链接就跳转到新的页面，或者提交一个表单。但是现在，页面和 App 更强调每一个操作都给用户反馈，所以会增加一些动画效果。有些是为了视觉更炫，有些是为了给用户操作反馈。但是这些动画是很难在设计稿中体现的，如果采用传统的开发流程，就需要设计师对每一个动画都作出描述和适配，工程师去实现它。或者需要设计师跟工程师长时间地沟通和交流，以确定每一个动画都能实现。

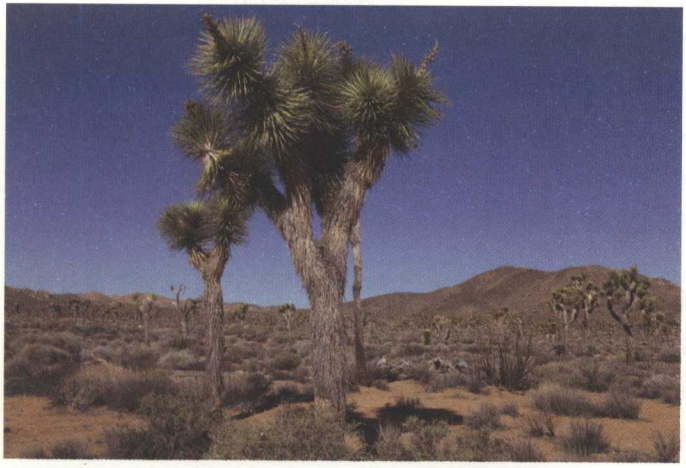
我们采取的优化方案是，让有一定设计功底的前端工程师，主动提出自己对动画的见解，比如在 CSS 动画中使用贝塞尔曲线，或者用一些有弹性的动画效果，做出效果之后再反馈给设计师去确认。优化后的流程十分高效。

在我翻译的《响应式 Web 设计全流程解析》一书中，作者提出了一种新的开发响应式 Web 界面的方式，就是让设计师不再使用 Photoshop 创建固定的设计稿，而使用 HTML 和 CSS 等 Web 技术来创建动态的设计稿。让设计师学习 Web 技术，这种方式有点激进，我在实际推行的时候感觉并不好用。让工程师学习一些设计理念和设计原则，然后反过来让设计师确认，就相对简单一些，我们正在努力这样做。现在我们在招聘的时候也会把设计理念作为应聘者一个有效的加分项。

设计基础

在 Web 设计领域，我们经历了早期苹果带来的拟物风格、Web 2.0 设计风格，以及微软扁平风格的演变，但是在支撑这些设计风格的设计理论一直没有改变。

在美国西部加利福利亚州和犹他州，有很多这样的树。



约书亚树。

这种树虽然造型奇特，但是随处可见，所以它也没有太吸引人的注意。它在美国人的生活中随处可见，但一般人也叫不出名字。一次偶然的机会，Robin Williams 了解到这种树叫做“约书亚树”，从此以后，每当她看到这种植物，就会知道它是约书亚树。

我跟妻子在去美国旅行的时候，特别去了加利福利亚州的约书亚树国家公园，在这个广阔的国家公园里长满了大量的约书亚树。回国之后，我在写这本书的时候，妻子从屏幕上看到约书亚树，就马上脱口而出：“约书亚树！”

这件事告诉我们，当我们知道一个“东西”的名字，就会在看到这个“东西”的时候，立马意识到它。但是如果我们不知道它的名字，可能一辈子也不

会认识它。这就是约书亚树原理。

关于设计的理论也是如此，普通人见过足够多的报纸、新闻、广告、传单、名片之后，其实潜意识中已经有能力判断哪些是“美”的，哪些是“丑”的，哪些是“专业”的，哪些是“业余”的。虽然我们不清楚某个设计为什么美，也不知道怎么调整使之更美，但我们往往知道这个设计很好，那个设计怪怪的。这种设计感，就是我们身边的“约书亚树”，我们知道它是我家门口的树，但是不知道它叫什么名字，也不会特别去关注它。

饶了这么大一个弯子，我就是为了推荐这本 Robin Williams 的《写给大家看的设计书》，这本书条理清晰，简单易读。一个周末的下午，或者每天半小时，持续一周就可以读完。但它的理论会如此深刻地停留在您的脑海里，每次看到不符合这些理论的设计，对应的理论就会迸发出来。

设计的四大基本理论是：亲密性、对齐、重复、对比。距离第一次读这本书已经过去 5 年，但是这些理论经常在生活中不由自主地跳出来。当我走在路上，接到一份广告宣传单的时候，当我看到路边一个广告板的时候，当我看到值得推敲的设计稿和页面不自然的时候，我就马上知道，设计师违背了基本的设计理论。

这 4 个设计理论分别是什么意思呢?

- **亲密：**关系亲密的元素要放在一起，关系疏远的元素则要分开。位置的亲密性直接表现出意义的相关性。
- **对齐：**左对齐、右对齐、上对齐、下对齐。斜线对齐比较简单，居中对齐很难处理，新手不要尝试。
- **重复：**视觉上使用重复的图形和元素、线条和颜色等。比如 QQ 空间重复使用的黄色跟黑色、微信的绿色、京东的红色等。
- **对比：**如果两个元素（的大小或者颜色）不一样，就让它完全不一样，产生视觉冲击力。

有些设计理念也会影响到我写代码和设计网站架构的方式。

比如设计的“重复”性会更加鼓励我复用已有的代码和图片，提醒我全站要使用同一个加载图标。

比如“对比”，当我看到设计稿中有很多颜色相似的元素时，我会假定认为它们是同一种颜色，如果有微小差别，那肯定是设计师随便选了一个类似的。因为如果两个紫色看上去差不多，但是我分别去取色的话，就会在代码中存在两个变量，要修改的时候就麻烦了，而且视觉上可能也会有微小误差。设计师应该是清楚这些设计原理的，所以如果他的原意是使用两个不同的颜色，那就应该用完全不同的颜色来设计。

所以“设计感”其实是“科学”而不是“艺术”。

这些只是理论或者“规则”。规则总是可以被打破的，但是前提是要熟练掌握这些规则。在没有掌握规则之前，请遵循规则。

设计工具

互联网公司对工程师掌握设计工具的要求不会像设计师那么高。设计工具主要分为两种使用场景，一种是处理，一种是创作。

处理是指处理视觉设计师交付的设计稿，对于 Web 工程师和 App 工程师等直接跟界面打交道的工程师，至少需要熟练使用 Photoshop 和 Fireworks 工具来分离图层、导出格式，批量切图等。这属于基本的技术要求，跟设计感关系不大。

创作是指利用自己本身的技术优势创作一些视觉设计师不擅长的产出，比如响应式页面、App 交互和动画，或者 CSS3 动画和滚动视差。

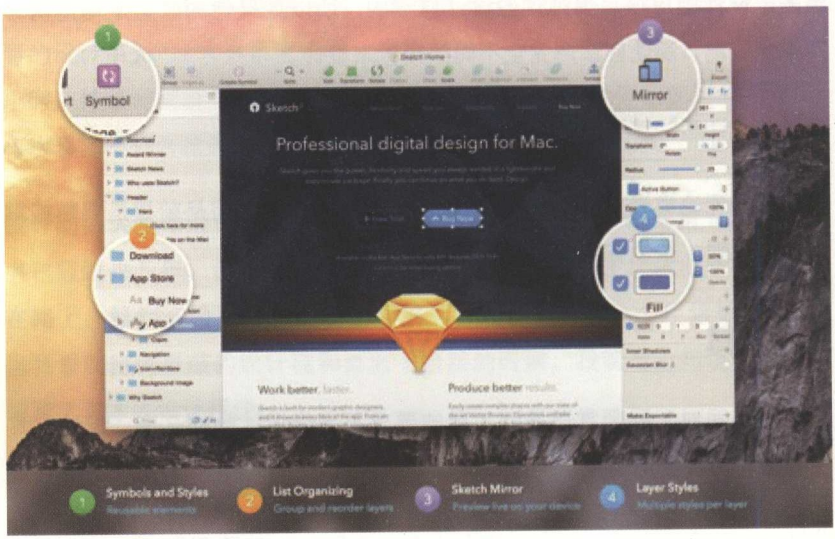
下面是一些常用工具的介绍。

- Axure

梳理完成全部产品信息架构和功能。Axure 以页面为最小单位，目的是输出整个网站的页面脉络，以及每个页面的交互框架。

- Sketch

Sketch 是一个轻量的设计工具，能够基于之前的线框图增强设计感，具象之前的产品。开发 Sketch 本意是帮助设计者能够短平快地画出精致的用户界面，因为 Photoshop 中有大量图片处理等界面设计不需要的功能，过于臃肿。



使用 Sketch 可以方便地制作 UI 设计图，图片来自 Sketch 官网。

- Quartz Composer

QC 是一款图形化的编程工具，专门用来生成各种复杂和细腻的动态视觉效果，以及可交互的界面原型。因为 QC 本质上是编程工具，所以一些视觉设计师不太喜欢这种方式，作为全栈工程师的您可以承担这个工作。

- 代码

代码往往是实现产品的最后一步，不过我们也可以把设计跟代码结合起来，

敏捷开发。比如，用 HTML/CSS 代码来实现响应式的高保真设计稿，您可以把静态页面直接托管在云服务器上，看看用户的反应，或者拿给老板和客户看看他们的意见，根据评论来优化设计。

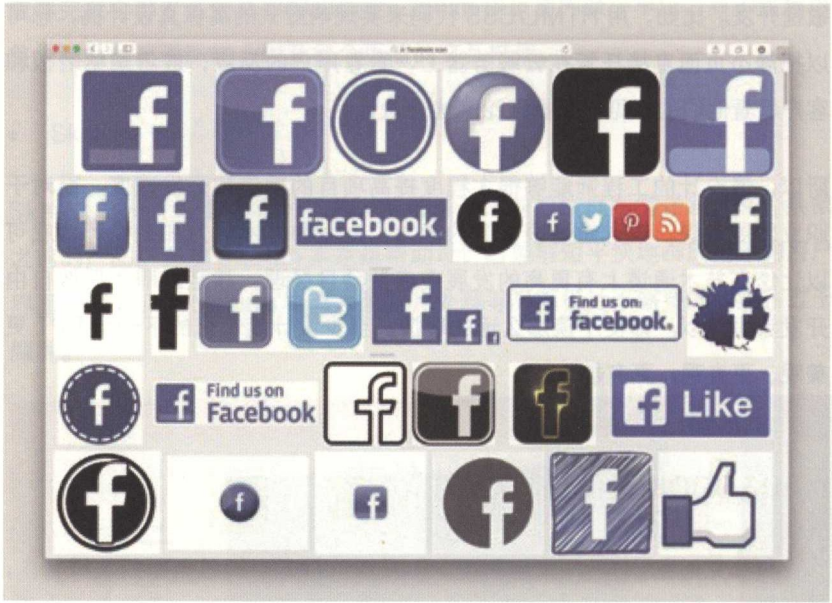
所以，懂设计的工程师能够很大程度提高项目的质量和开发速度。而对于职业发展来说，设计也是很重要的能力，全面的技术能力加上设计功底可以让您在技术通道上有更高的发展或者做项目经理，甚至辞职之后做自由开发者也可以。即使不谈那么远的事情，至少设计能让您的下一个 PPT 更美观、更直观、更有趣。

Facebook 的品牌设计故事

马克·扎克伯克于 2004 年创建了 Facebook 网站。经过不断的发展壮大，在 2010 年 3 月，Facebook 的访问人数已经超过 Google，成为全美用户访问量最大的网站。

之前说到 Facebook 一直坚持“Move fast and break things”，技术方面体现在代码写得粗野豪放，据内部工程师说，甚至有时候他们以研究高科技把服务器搞挂为荣。Facebook 大量采用“LAMP 架构”，所以在用户量越来越大的时候，他们开发了“HipHop for PHP”，编译 PHP，比纯 PHP 脚本的性能提高了一倍。这也能看出他们在技术上追求无快不破、不追求完美的文化。如果项目能成功，再来优化技术。

不过在公司整体品牌设计上面可能就没那么容易变革。说到 Facebook 的设计，因为早期公司并没有特别注重品牌设计风格的统一，对外也没有一份官方的设计指导和资源下载，所以在 2012 年的时候，搜索“Facebook icon”可以看到大量第三方网站设计的图标。不同的图标有不同的英文字体、颜色、高光、边框、立体感……不得不说，这些图标出现在各种网站的时候，对于网站（公司）品牌是一种伤害。此外，您可能还会注意到，不同图标的 F 字母与图标的边距各不一致，有些图标下面有一条蓝色下划线，有的图标则没有。



Facebook 早期图标比较混乱，图片来自 Google 搜索。

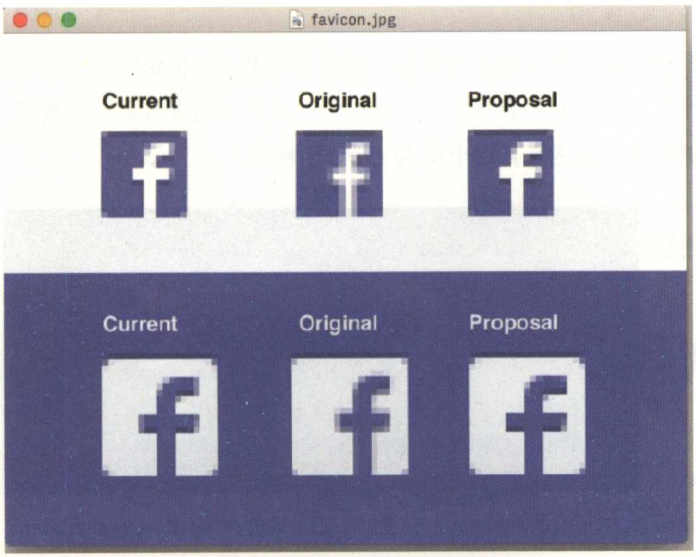
所以，当 Facebook 的用户越来越多的时候，Facebook 的模拟研究室设计师 Ben Barry 开始重新设计整体品牌。

除了 logo 之外，网站另一个重要的品牌标识就是网页图标。[1] 首先，设计师去掉了 Web2.0 风格严重的蓝色下划线，调整了字体和背景色的质感，整体看上去更加清爽和现代化。

新的 logo 调整了字形和光影，并去掉了下划线。

1 网页图标（favicon）是与某个网站或网页相关联的图标。网站设计者可以用多种方式创建这种图标，而目前也有很多网页浏览器支持此功能。浏览器可以将网页图标显示于浏览器的地址栏中，也可置于书签列表的网站名前，还可以放在标签式浏览界面中的页标题前。

此外不得不考虑的一个问题是，网页图标并不仅仅出现在浏览器的书签栏或者地址栏，它还会出现在高级浏览器的某些地方，如 IE9 的 pin 栏，会在工具栏生成一个网页的快捷方式，尺寸是 32x32 像素，所以如果使用 16x16 像素的图片，会有一个白边，很难看。但实际上这样操作的用户并不多，除非用户是网站的铁杆粉丝，并且习惯使用系统的这一操作。另一种情况是，视网膜屏幕上的浏览器在标签栏显示，会需要 32x32 像素的图片。



网页图标的优化需要考虑 16x16 像素分辨率，图片来自 officeofbenbarry.com。

为了让图标在缩小为 16x16 像素和 32x32 像素时保持清晰，需要做一些微调。从上图可以看出，原始图片的“f”字形周围有明显的半透明虚边，而优化过后就清晰多了，只用半透明制造曲线和圆角。

网页图标通常在 HTML 的头部设置，代码如下：

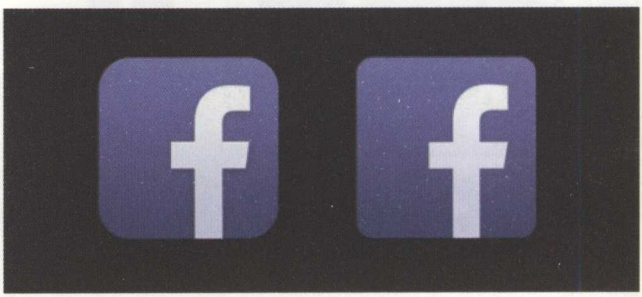
```
<link rel='icon' href='/favicon.png' type='image/png' />
```

对于网页图标的图片格式，不同的系统和浏览器有不一样的支持度。我推荐使用 ico 格式，因为 ico 格式可以在一个文件中同时设置 16x16 像素和

32x32 像素的图片，根据需要来显示。ico 格式的生成也非常简单，使用 Photoshop，将一个 32x32 像素的普通图片直接保存为 ico，就可以生成一个拥有两个图层（32x32 像素图层，16x16 像素图层）的 ico 格式文件。ico 文件里面图形的默认位深为 32 位（RGBA，代表支持半透明）。

如果希望 16x16 像素图层不仅仅是 32x32 像素图层的缩小（比如像 Facebook 的设计师一样进行一些像素级的微调），可以通过专门的图标编辑器（比如 Iconworkshop）来单独编辑某个尺寸的图层。

修改完网页图标，还有两个重要的图标，就是 iOS 和 Android 上的应用程序图标。两个平台有不同的圆角大小，所以设计师又分别进行了一些微调。您能分辨出哪个是 iOS 图标，哪个是 Android 图标吗？



Facebook 的 iOS 图标和 Android 图标，图片来自 officeofbenbarry.com。

经过一些调整，现在形成了最基本场景下的风格设定。

为什么说是最基本场景呢？因为还要考虑背景色是深色、浅色、反转色甚至单色的时候用什么样的形象。这些统一形象背后的理论基础就是“重复”。

理解设计原则对全栈工程师有什么帮助呢？帮助很大。您可以在制作网页图标、网站 logo、外网挂件的时候考虑素材的一致性。因为最终这些资源和代码都需要您提交，所以您清楚自己输出的图片有哪些不一致。您可以提醒设计师补充各种情况下（深色、浅色、反转色、单色）网站 logo 的设计，并给出设计指南；您可以提醒设计师在视网膜屏幕上给出单独的图形设计。

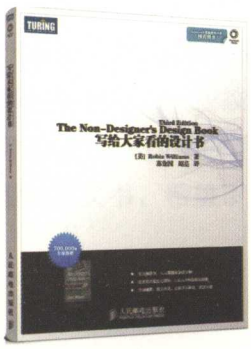
视觉设计师在更新设计版本的时候可能会有遗漏，您也可以及时提醒他。

延伸阅读

1. 《响应式 Web 设计全流程解析》（美）Stephen Hay，人民邮电出版社

2. 《写给大家看的设计书（第 3 版）》（美）Robin Williams，人民邮电出版社

3. 《写给大家看的设计书：实例与创意》（美）Robin Williams，人民邮电出版社



全栈思维

2012 年，我组织腾讯 ISUX 设计团队的十来个人共同翻译了《众妙之门》(The Smashing Book) 一书，因为对于互联网工程师的我们，这是第一次跨界做好玩的事情。所以，这本书出版后，我在腾讯大厦的一个兴趣分享会上，分享了图书出版背后的一些故事。

在会后，两个刚毕业的小伙子联系到我，说他们很有兴趣翻译各种类型的英文书，希望下次有机会也来参与。

有兴趣就够了吗

学一点管理

沟通：被忽视的竞争力

有兴趣就够了吗

最近，我被一本书的翻译稿搞得每天睡眠不足，所以非常清楚翻译是个很耗费时间和脑力的工作。阮一峰在翻译完《More Joel on Software》之后说道：

> 话说今年 8 月份，我翻译完《More Joel on Software》，已经精疲力竭，对这种通过长时间击键，将英语改写为汉语的廉价体力 + 脑力劳动深恶痛绝，再也不想干了。

没错，当时的我就是抱着再也不想干了的想法跟他们说："如果出版社有更多的需求，而您们有时间和兴趣的话，我就把任务转给您们。但我是没有时间再去做这种工作了。"

后来出版社编辑跟我说，有一本响应式设计流程方面的书。我初步阅读之后，觉得这个题材很新颖，就转给这两个年轻人。他们评估之后觉得两个月应该能完成，我也答应到时候帮他们做校对。

但结果是，他们出于各种各样的理由（搬家啦、加班啦、参加其他活动啦）延期交稿。延期一周了也不主动告知我原因，所有的沟通都是我主动去问，然后才告诉我延期了，保证下个星期交稿。后来又催了好几次却一拖再拖，耗费 4 个月的时间也没能完成各自预期的 50% 的工作量。我只好拉了其他人来帮忙翻译剩下的部分。

而他们完成的部分也只能算勉强及格，错译、漏译的情况常常出现，可能存在一些能力问题，但他们给我的感觉却是根本就不上心。

大半年的时间，我都在收拾这个烂摊子，虽然这个项目本不该由我来主导，我只是作为一个中间人来传达这个项目。但我知道最终出版的书上会写上我的名字，我也没有办法让自己撒手不管，我告诉自己一定要投入时间去做好它。

如果想拖延一件事，或者不想做一件事，总是能找到理由。懒惰的终极原

因就是您想逃避这件事。对于所有刚开始工作的年轻人，我告诉您们：

老板给您任务，根本不关心您有什么理由，只关心您完成没有。

扯一些理由（特别是私人的理由）根本就是不专业的态度。不要给自己没能完成的事情找任何理由，而要以诚恳的态度说明当前进度，以及未来是否能如期完成目标，如果不能，就直接说出来。这样老板可以对进度有所了解和预期。

在职场中，我们评估一个人并不是根据他的能力，而是他能承担的责任。无论项目成功或者失败，准时完成或者延误，如果有一个人能主动汇报和负责，拍着胸脯说“我负责”，那他就应该是项目的主导和骨干，而不论年龄和资历。

同样，我们最终评估一个人是否应该升职加薪，不是看他的能力，也不是看他投入的时间，更不是看兴趣，只是看他的责任。他在一个个项目中都承担起了责任，就是一个有责任心的人，就能委以重任。

有人认为兴趣是成功的老师，无法完成某些事情是因为没有兴趣。其实我认为耐心是一种能力，有些人天生缺乏这种能力。在能力不足、困难重重的时候，唯有投入大量的时间才能保住这珍贵的信任。

新人没经验、知识不丰富，这都可以理解，但是以此为理由输出不合格的产品，那就是自己的问题。我在实习的时候，通宵睡在公司都经历过。

您有没有想着把您的产品和您的名字联系起来

我们都希望把自己的名字同成功的项目和作品联系起来，让人们谈到这个项目的时候，自己也很自豪。但是在这个项目还在孵化的时候，您是否也有把自己的名字跟项目联系起来的觉悟?

对于翻译图书，保证质量并不是其他人的责任，不是审校者的责任，不是测试的责任，而是您的责任。这就是全栈工程师精神。

您是否珍惜您的时间？其实我们很缺乏对时间的管理，十几年的学生生涯中，每一个阶段、每一周、每一节课……该干什么是很固定的，任务持续的时间也是老师安排的，我认为“木偶戏”可以很形象地描述这种现状。而在踏入工作之后，所有任务一下子变得不是很明确，完成的时间也需要自己评估。

到目前为止，我有5年多的工作经验（包括实习），从最早纯粹的完成任务，到现在投入越来越多的时间在团队管理上，所以对于时间管理，我有一些自己的经验。

有些人觉得公司新人比较忙是合理的，因为活儿都丢给新人去干，等渐渐当上管理层，就闲散了，因为可以把活儿丢给别人干了嘛！事实是截然相反的，管理者要做的事情比执行者更多，但是因为管理者能管理好自己的时间，做事讲究优先级，懂得授权和请求他人帮助，所以能避免“瞎忙”。让自己“不忙”，可是一个技术活。

所以在工作中，珍惜自己的时间尤为重要。管理者第一个需要练习的就是管理自己的时间。

学一点管理

虽然前面都在批评别人，不过在这个项目中，我是事实上的管理者，所以我也要负责任。在这个项目中，先姑且把这两个年轻人称为我的“下属”。

- **没有在最开始做出合理的时间评估**

下属对于时间的评估，我应该质疑。我应该猜测他们的能力可能达不到要求，或者他们对自己可以投入的时间预期过于乐观了。

- **没有根据人员的强项来安排任务**

我不了解下属的强项，在他们没有足够经验的前提下就完全授权，项目的

失败几乎是必然的。除了授予工作、任务和责任以外，还要授予知识和资源。作为有经验者，我完全没有给他们提供帮助，这是我失职的地方。

- **没有唤起他们对项目成功的渴望**

如果您想让团队一起造一艘大船，您不要告诉他们如何建造一艘船，而应该唤起他们对大海的渴望。领导者的工作本质是对内销售，销售的内容就是团队的目标。我没有唤起他们对完成这个项目的渴望，就匆匆开始项目了。

- **缺乏沟通**

对于一个预计 2 个月的中期项目，理想的沟通频率是每周沟通至少两次，如果两周才沟通一次，失败也是可以预期的。任何管理者，安排了任务之后，如果长期不询问进度，就代表不关注。上司都不关注，下属自然就不会上心了。

不是每个人都有足够的自律和积极性。虽然作为全栈工程师，我们的学习目标一直是提升个人的技术能力。但是在组织中工作，并不需要特别强的个人能力或者天赋、更需要的是稳扎稳打、虚心学习，不要害怕批评，而应真诚沟通，珍惜每一次机会，完成每一个承诺。

好的管理者能让平凡的员工做不平凡的事

在明星 IT 公司 Netflix 的 PPT《Netflix 文化：自由与责任》中，他们阐述了自己的企业文化和招聘风格：

> 用市场最高价格雇用高效能人士组成团队。

对于这个观点，我部分赞同。我的观点是，对于脑力劳动者的自我修养，这些看法值得我们学习，我们都要让自己成为价格最高的专业人士。但是对于管理者，这一文化纯属心灵鸡汤，并不太有可操作性，您总会看到每个成员不完美的地方，要把他们都开除吗？

高效能的管理者并不奢求完美的人才，他能让平凡的人成就不平凡的事业。

在德鲁克的经典管理书《卓有成效的管理者》中，他建议管理者学会这么一种创建组织的方式：若某人在某一方面具有特长，就要让他充分发挥这一特长；而不应该期望另寻万能的天才来达成绩效。这本书堪称提高脑力劳动者专业度的圣经。

值得在最开始就明确的是，本书的“管理者”并不是以他有没有下属而定，而是看他是否能做决策，并承担起做贡献的责任。本书的英文名《The Effiective Executive》中 Executive 既有传统的“管理人员”的意思，又有负责执行和判断的“执行者”的意思。所以管理者泛指工作者、经理人员和专业人员，包括工程师。由于其职位和知识，他们必须在工作中做出影响整体绩效和成果的决策。

有天生的管理者吗？我在上学的时候觉得自己这辈子应该做不了管理类的工作，因为我很内向，对人不热情，永远不会自来熟。我没有办法很快记住一个人的名字，也有点脸盲，在团队中也不是会说很多话的人。我喜欢专注做事情，我以为我会一辈子搞技术。

但是我现在必须做管理，怎么办？

好在管理者的特质不是天生的。假如卓有成效是一种天赋，那就糟糕了，今天世界上那么多大型组织，要去哪里找那么多有管理天赋的人呢？事实上，卓有成效的管理者并无性格上的共性，唯一的共性就是他们要经过长期的训练，从而养成 5 个思维上的习惯。

这 5 个思维习惯是《卓有成效的管理者》的核心，环环相扣，非常经典。

- 有效的管理者知道他们的时间用在什么地方。
- 有效的管理者重视对外界的贡献。
- 有效的管理者善于利用长处，包括自己的长处、上司的长处、同事的长

处和下属的长处。

- 有效的管理者集中精力于少数重要的领域，在这少数重要的领域中，如果能有优秀的绩效就可以产生卓越的成果。
- 最后，有效的管理者必须善于做有效的决策。

每一条都会花一章的篇幅来展开说明，每一章都有些让我醍醐灌顶的部分。比如“有效的管理者重视对外的贡献”。

重视贡献，才能使管理者的注意力不为其本身的专长所限，不为其本身的技术所限，不为其本身所属的部门所限，才能看到整体的绩效，同时也才能使他更重视外部世界。

根据员工特质来授权

对于“授权”的意义和目的，很多人都有误解。授权并不是为了推脱工作，而是为了把精力更集中地放在自己想推动的重要项目上，而不是被一些杂事或者自己不擅长的事困扰，浪费精力和时间。

授权给下级员工的时候，主要是要考虑员工的意愿和能力，沟通的时候顾及感情，不要让对方觉得您在拿职位压他。

对于全栈工程师型的领导，可能什么都想自己做。而自己的下属，要么有意愿但是能力不够，要么有能力但是意愿不够。

对于有意愿但是能力不够的，要教会他做事的方法，提升其学习能力，这部分员工是很有潜力的，往往是年轻毕业生居多。

对于有能力但是意愿不够的，就要多进行情感上的沟通，看看对方情绪不够积极的原因是什么，对待这部分员工会困难一点，需要领导有很强的沟通能力。

既有意愿又有能力的员工，比例比较小，如果发现了，要大力培养，作为后备人才来储备。

对于既没有能力也没有意愿的员工，尽早开除比较好。这种员工就像蛀虫，会腐蚀一个团队。

沟通：被忽视的竞争力

一个事实是，程序员的沟通能力所带来的价值被大大低估了。

在我们的招聘流程中，技术能力过关，但是因为沟通能力这一项不过关，而直接被拒绝的面试者比例还比较高。但是为了避免不必要的争议，大企业的 HR 往往不会把拒绝的原因传达给应聘者，所以对方也不知道自己为什么被拒绝。这是通用做法，在目前所有行业的招聘中，企业都没有义务主动告知应聘者被拒绝的原因。这样做主要是为了避免无修止的争论。“我们觉得这个职位需要沟通能力强一点的人”。“我们认为您的沟通能力还不够强。”您可能会收后如下回复：“其实我的沟通能力很强，但是面试中太紧张了，所以没有表现出来。”

您看，沟通能力的评判往往是非常微妙和主观的，并没有一份考题能证明您的沟通能力好或者差，只是面试官能根据自己的判断来决定。为了避免无休止的争论，所以干脆不告诉拒绝原因是最好的。不过，其实应聘者也可以跟面试官搞好关系，然后咨询改进意见，不要问“我被拒绝的原因是什么？”而是问“我在面试中的表现怎么样？”这样往往面试官会愿意给出一些建议。

这跟对心仪的女生表白是一个道理，对方基本上会拿这些原因拒绝您：“您是个好人”“我们不太合适”“我已经有男朋友了”“我现在还没准备好”。她绝不会说出“其实您太穷了、太难看”这样的原因。

所以，企业对程序员的沟通能力要求很高，在能力图谱中权重很大，这一信息往往并没有明确被传达出来。

另一个原因是，传统行业、四格漫画以及微博段子手对程序员这一行的标签就是“木讷”“不善言辞”“格格不入”“怪僻”等，甚至某些程序员自己也把这些标签作为自己的优点，诸如“如此不善言辞的我也能得到这份还不错的工作，靠的全是实实在在的技术能力啊，对于这样的技术能力表示骄傲自豪，所以我不需要提高沟通能力。”

但是这样的错觉维系不了多久，因为入职一段时间您就会发现身边的人，甚至后进生，得到更好的项目以及更快的晋升机会。善于反思的人会开始想自己的不足，心怀负能量的人就直接抱怨“马屁精，一门心思讨好领导，才获得了这么多机会吧，我还是跳槽好了。”

沟通是软技能

有些人认为程序员是跟计算机打交道，来把自己的思想注入到程序之中，片面理解“code talks”，其实不然。

图片来自《编程的本质》。

这里引用 Ruby 之父松本行弘在《编程的本质》中的一句话：

> 尽管看上去是和计算机打交道的工作，但实际上编程的对象还是人类，因此这是个非常“有人情味”的工作。
>
> ——松本行弘

因为编程的对象是人类，所以欠缺沟通能力的话，就可能造成项目方向错误、进度延期甚至完全失败。

我认为良好的沟通是：“针对目标听众”“有方法地”“表达自己的想法”。接下来我分别谈谈这 3 个话题。

针对目标听众

我们每天会跟很多人聊天、网络聊天、E-mail……并不是人人都需要使用特定的方法去沟通，比如您的家人。但是如果您太太对着您絮絮叨叨毫无重点地讲一晚上，您也会抓狂吧。而如果是对于一些有利益关系的听众，特别是职场上的伙伴，就一定要掌握沟通方法。

有一次我想推动一个跨团队的优化项目，是使用新的 HTML5 技术进行性能优化，其中的数据方案和取证需要对 CDN 后台数据进行挖掘分析。负责 CDN 后台是另一个集团负责的，而且整个项目中有一大半的代码量都不是我能完成的，所以我的主要精力放在推动各个部门的其他岗位的同事来协作。

腾讯公司有两三万人，假如您收到从另外一个集团或者部门的同事发过来的协作请求，而且对方跟您是平级的员工。为什么您要帮助他？

从功利的角度来讲，这件事情要对我自己有利我才去做。

佛家有一个词叫“度己度人”，就是在帮助别人的过程中，其实也是帮助自己。所以反过来想，作为需求的请求方，最开始就得找到那个很关键的人，对于他来说，帮助您对他是很有好处的。也就是说他能把这件事当作自己的关键任务（KPI）。如果您的要求对于他人纯属累赘，那么他人自然不愿

意帮您了，任您多么会沟通，最终都不管用。

所以，授权给平级同事的时候，**最好的方法就是诉诸对方的利益**。如果一件事情可以对双方的 KPI 都有好处，那么对方也愿意帮您一起分担这个任务。如果您把不擅长的事情授权给对方，而作为交换，能给对方一些资源，那也是诉诸利益的一个好方法。

其次的方法就是把问题上升到上级领导，让上级领导安排资源，但是这种方法不能经常用，否则上司会认为您不会主动解决问题，只会提出问题。被授权的那一方也觉得您在拿领导压制他，可能会存在负面情绪。通过跟对方主动沟通，并且在邮件和 RTX 群中多赞赏对方，就能唤起对方的积极性。

针对同事的沟通目标往往是请求对方帮助。需要对方做事，原则就是“度己度人”，那么针对上司呢？有以下几种情况。

- 汇报：求上司表扬。
- 请求：需要上司做事。
- 询问：需要知道上司对于某项任务的要求。

对于这些特定的目标，有不同的方法，但是有一点是肯定的：上司时间非常宝贵，您需要在 20 秒之内让他知道您的目标。

有方法

有的读者可能已经看出来了，这就是麦肯锡的金字塔原理。

金字塔原则就是，任何事情都可以归纳出一个中心论点，而此中心论点可由 3 至 7 个论据支持，这些一级论据本身也可以是论点，被二级的 3 至 7 个论据支持，如此延伸，状如金字塔。

我再次强调，使用金字塔方法的前提是，您得有一个中心目标。不能是两个，更多更不行，只能是一个。

金字塔原则不止在当面沟通中有用武之地，在邮件和 IM 中更加适用。在邮件中有一个更有意思的术语，可以是金字塔原理的改良版，就是“战地记者原则”。假设您是一个战地记者，在战场发送一封邮件回报社，这时候您怎么写这封邮件？

枪林弹雨，战火硝烟，这时候没人关心您的文法和词汇，您的目标是在一分钟内说明当前的情况，不超过三句话，然后发表出去。

表达自己的想法

我们现在了解了沟通的听众——比如上司。

也有了沟通的目标——汇报项目如期上线，然后要做的就是表达自己的想法了，敲开上司的办公室，直接说：

> xx 项目已经完成 / 上线。获得了成绩 / 收入 / 优化效果等。我们的活跃用户数量提升了 xxx。我们使用了 xxx 技术，能够用同样的成本，服务更多用户。

这是汇报，或者叫“求表扬”，因为不需要上司做任何事情，他听了很高兴。但是五分钟后您话锋一转：

> 但是我们现在的三个人，A 投入到 xxx 模块中，B 投入到 xxx 功能的开发中，现在每天都在加班，PM 设定的第二期时间点是 xxx……

就让上司有点恼火了，因为说了五分钟才到重点，他可能会生气地问您：“所以我需要做什么？”比较好的说法是：

> xxx 项目第一期已经上线，获得的成绩是 xxx，目前进展顺利。但是基于现在的项目进度和我们的人手安排，第二期有延期的风险。所以请老板评估下，我们现在有几种方案：一、暂缓其他项目，抽调人手到这个项目；二、缩小二期的范围（即砍掉一些功能）；三、延期上线时间。关于暂缓项目……关于缩小范围……关于延期上线时间……

重点是，因为上司时间很紧迫，所以要在第一句话就告诉上司您的目的，不要掖着藏着，既想请求上司的帮助，又要求表扬。

无论您的想法多么复杂，已经进行了多么复杂的推演，最终您要沟通的时候，要注意两点：一是让自己要表达的立场足够简单；二是围绕着这唯一的立场去沟通。

示例：谈谈 PPT

我这几年有不少分享，但都比较小型，一般只有十几个听众，多的时候也不超过一百。听众群体比较多变，有的时候都是大家跟我一样的前端设计师，也有时候是交互设计师和视觉设计师，还有时候是公司各种岗位的同事。

我的演讲能力也逐渐得到提高，最开始是念稿子，经过这些年的学习和总结，现在可以充当一个业余的相声演员，也有一些经验可以谈谈。

演讲者往往会进入一个误区，他们假设听众对自己的分享很有兴趣。其实在公司内的一些分享，很多人来听往往是忙里偷闲，或者碍于情面来捧场，并不会对话题本身有特别浓厚的兴趣。

所以我的第一个建议是，用最悲观的假设来思考听众的目标。您应该假设听众对您分享的技术完全没兴趣。以这一目标来进行的话，您可能会有自己的答案。

其次，您要有一个鲜明的立场，并让这个立场简单，并且演讲始终围绕这个立场。与金字塔不同，您不用一开始就说出这个立场，您只要清楚这个立场就行，然后尽量让演讲的氛围轻松一些。此外，在演讲时还要掌握以下技巧。

- 不要有太多文字

听众的注意力只能放在一个地方，要么是幻灯片文字，要么是您自己。除

非您不想让大家关注您，否则幻灯片中的文字应该尽量少一些，因为幻灯片只是辅助。

- **设定进度**

让听众知道自己在哪里，比如一开始就告诉听众您的演讲时间，或者在幻灯片中设置进度表。听众注意力只能坚持十分钟，所以不要太久。

- **对待错误：放松**

跟自己预想的不一样？放松，没人知道您犯了错。记住，您怎么对待错误，听众就怎么对待错误。您觉得这是世界末日，它就是世界末日，您觉得可以跳过，听众就觉得可以跳过。

- **有条件的话，录像并对比提高**

不要相信别人的评价，碍于面子，别人不会提出不好的评价。

大概就是这些，此外的建议就是多加练习，把自己推出去，不要待在舒适区里不出来。

内向性格的竞争力

我的性格是比较内向的。“内向”两个字都是很中立的，但是放在一起似乎就成了贬义词。我颇不以为然，所谓自我提升，首先是接纳自我，而不是贬低自我的性格。这种“有的性格比其他性格更好”的消极心理学，除了贬低人格，增强自卑，是没什么益处的。就好像左撇子跟右撇子没有孰优孰劣，但是家长就偏要纠正，让孩子苦不堪言。

我是内向性格，这让我有很好的学习能力和不错的社会意识。

毕业这么多年，我从来没有停止学习，技术、管理、英文……因为内向性格的人更能够沉下心来读书、思考，做一些安静的事情。这样的事情让外向性格的人哈欠连连。

对于内向性格的人来说，跟人交往是对精力的消耗，但并不代表情商低。情商低的人不能察觉到自己和他人的情绪波动，也许很外向，但是讨人厌恶。内向但是情商高的人，能够感知周围人的情绪，并敏锐地捕捉到周围发生的事情。

很多工程师的性格都是内向的，相信许多人都有点害怕在公众面前讲话，我其实也是。不过经过这么多年的职场学习和实践，我的沟通能力得到了很大的提高。

其实，能灵活自如地适应各种社交场合的内向者往往有很强的自我意识、社会意识、自我管理和关系管理能力，加上自我学习能力，是很强的一种性格特征。

自我意识，是指您能够精准地觉察自己的情绪波动。

社会意识，是指您能感知周围人的情绪，并敏锐捕捉到周围发生的事情。

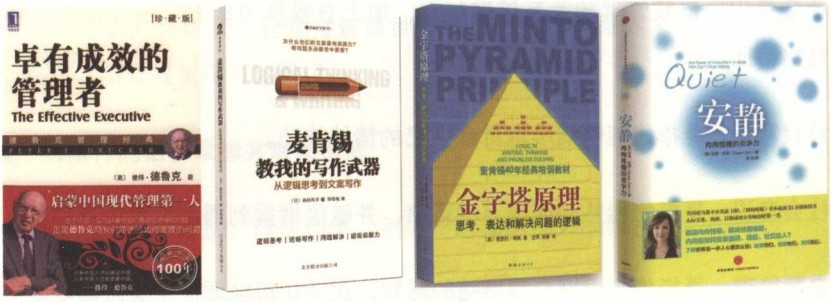
自我管理，是指您能根据自我情绪的感知，灵活积极地调控自身行为。

关系管理，是指您感知到周遭的“情绪场”之后，能够掌控自我情绪并把握他人情绪值，来让双方进行更好的互动。

记住，您的性格来自您的基因，有其独特的价值，只要充分利用自己天赋，就能得到您想要的。不要为了迎合主流社会，而伪装成一个外向、热情、合群的人。

延伸阅读

1. 《卓有成效的管理者》（美）彼得·德鲁克，机械工业出版社
2. 《麦肯锡教我的写作武器》（日）高杉尚孝，北京联合出版公司·后浪出版公司
3. 《金字塔原理》（美）巴巴拉·明托，民主与建设出版社
4. 《安静：内向性格的竞争力》（美）苏珊·凯恩，中信出版社



后记

拿着手中第一版图书样本，我心里有诸多感慨。

本书已经进入到最终校对阶段，可以认为这个项目已经 99% 接近完成。用 Web 开发的术语来说，已经完成了测试，进入了“预发布”阶段。

写一本书、翻译一本书、开发一个 Web 网站、开发一个 iOS APP，甚至策划一次长途旅行……都是一个项目。项目的定义是：在一定的约束条件下（主要是限定时间、限定资源），具有明确目标的一次性任务。对于写书来说，只要有一台笔记本、一个记事本软件就可以开始了，最大的成本就是精力，约束条件是有限的时间。Web 产品完成之后，我只要按下按钮就可以直接发布给上亿用户，但是也没有定稿一本书这么紧张，我想了想，原因有两个。

- Web 产品的发布是可以回滚、迭代的。这意味着如果出了点问题，我可以马上撤销，回到上一个状态，或者在下一个版本中修复改进这个问题。但是图书在出版之后，如果发现问题，就只能记下来，在印刷下一版本时改进，之前的读者收不到更新。

- Web 产品有良好的功能拆分：在交付软件时代，微软要闷头开发一年，然后将软件放在光盘中分发给用户；但是在 Web 时代，几乎每一个版本都只涉及部分功能，通过小步快跑、快速迭代的方式优化产品，这样也比较好测试，风险也都在掌控中。

所以，出版一本书就像发布一个交付式软件，必须很慎重。好在互联网把人与人都更方便地连接了起来，也在持续地优化创作者和用户之间的管道。手机 App、Web 站点、图书都乘着互联网的疾风，开始互相借鉴。App 可以后台无打扰升级；Kindle 电子书可以随时更新，直接推送到读者的电子设备；Web 站点可以通过 Manifest 设置离线资源包，让用户可以离线访问站点。

对于全栈工程师来说，这是最好的时代，因为有了这些新的硬件、标准、平台、技术、协议、API、市场……我们的精力可以放在产品打磨上，而不用操心太多细枝末节。

这本书可以出版，不得不感谢责任编辑赵轩的支持与鼓励，是他的鼓励让我下定决心创作本书，是他的督促让我终于在 deadline 之后的两个月完成了书稿，是他的专业素养让本书看上去有模有样。

最后，感谢我的妻子，如果没有你的陪伴，我应该早就写完这本书了……

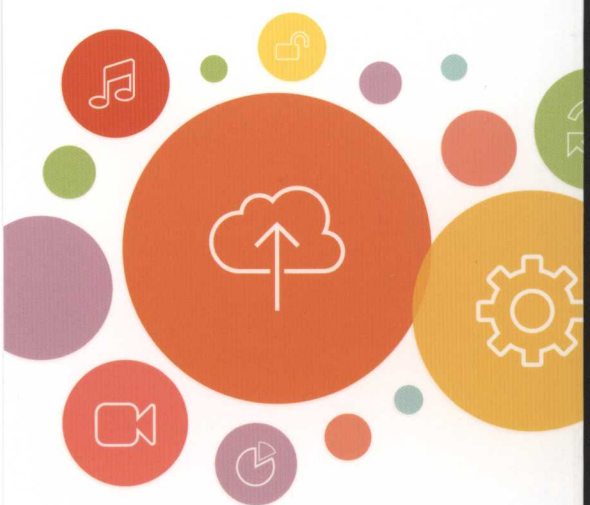
余　果

2015 年 7 月 21 日

全栈工程师正成为 IT 行业的新秀，无论是上市互联网公司还是创业公司，都对全栈工程师青睐有加。

本书作者是腾讯公司高级工程师，在前端、后端和APP开发方面都有丰富的经验，在本书中分享了全栈工程师的技能要求、核心竞争力、未来发展方向、对移动端的思考。除此之外，本书还详细记录了作者从零开始、学习成长的心路历程。

本书内容全面，客观务实，适合互联网行业新人、程序员，以及期待技术转型的从业者阅读参考。



《Web全栈工程师的自我修养》全面介绍了互联网开发的基本知识和行业状况。文笔流畅、通俗易懂，充满了有趣的小例子和作者的感悟，读来引人入胜。如果你想成为全栈工程师，它可以作为你踏进这个行业前的第一本书。

——阮一峰

《软件随想录》《黑客与画家》译者

《ECMAScript 6入门》《如何变得有思想》作者

每个人每年都会思考自己的成长，然后给自己制定一个目标。目标很重要，不过更重要的是完成目标的方法。余果在书中的切身感悟，相信不仅仅能让你了解如何成为“全栈工程师”，更能给你以启发，找到适合自己的成长方法论。

——神飞

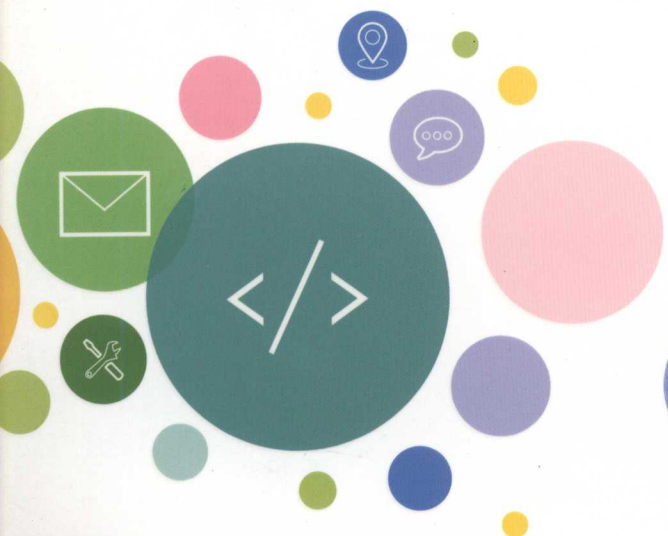
“前端观察”站长

微信设计中心高级UI工程师

在我看来，工程师称得起全栈，不止于学会从最后端到最前端的技术，还应该能够发现问题并运用“跨界”技术提供更好的方案。工程师好像天生拥有工具（代码），但如《代码大全》里常提到的，好方案在很多情况下更需要思考。作者跳出了代码进行思考，这对工程师来说是非常难得的。

——sofish

“饿了么”资深前端架构师



分类建议：计算机
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-39902-1
定价：49.00 元